JWST Pipeline Documentation

Release 1.13.5.dev0+g88dfde9.d20240226

jwst

Feb 26, 2024

GETTING STARTED

genindex | modindex

Welcome to the documentation for `jwst`. This package contains the Python software suite for the James Webb Space Telescope (JWST) calibration pipeline, which processes data from all JWST instruments by applying various corrections to produce science-ready, calibrated output products including fully calibrated individual exposures as well as high-level data products (mosaics, extracted spectra, etc.). The tools in this package allow users to run and configure the pipeline to custom process their JWST data. Additionally, the `jwst` package contains the interface to JWST datamodels, the recommended method of reading and writing JWST data files in Python.

If you have questions or concerns regarding the software, please contact the JWST Help desk at JWST Help Desk (https://jwsthelp.stsci.edu).

The following is a quickstart guide to installing and running and the latest version of `jwst`.

In short, the only setup required to run the JWST pipeline is to `pip` install the `jwst` package into a `conda` environment, and then to set correct environment variables for accessing reference files through CRDS. From there, the JWST pipeline can be *run in a Python session* or with the *command line interface*, and *JWST datamodels* and other pipeline utilities can be imported and used in a Python session.

1. Create a conda environment.

Python environments allow you to install different versions of packages and their dependencies and keep them isolated from one another. While there are several possible ways to achieve this (e.g `venv` (https://docs.python.org/3/library/venv.html#module-venv)), we will use `conda` in this example.

If you don't already have `conda`, please follow the install instructions (https://docs.conda.io/en/latest/miniconda.html).

To create a conda environment specifically for the latest stable release of `jwst` (in this example, called jwst_latest):

```
conda create --name jwst_latest python=3.10
```

This will create a new, (nearly) empty Python 3.10 environment in which you can install the `jwst` package.

2. Install jwst from PyPi

Once you have created your conda environment, make sure it is active by doing:

```
conda activate jwst_latest
```

To install the last stable release of `jwst`, and all its basic dependencies (e.g numpy, stcal):

```
pip install jwst
```

For detailed installation instructions, including how to install the development version of `jwst` from Github or how to install a previous released version, see the *Installation* page.

3. Set environment variables for Calibration References Data System (CRDS)

CRDS is the system that manages the reference files needed to run the pipeline. Inside the STScI network, the pipeline works with default CRDS setup with no modifications. To run the pipeline outside the STScI network, CRDS must be configured by setting two environment variables:

```
export CRDS_PATH=$HOME/crds_cache
export CRDS_SERVER_URL=https://jwst-crds.stsci.edu
```

The `CRDS_PATH` is the directory on your filesystem that contains your local CRDS cache, where reference files are accessed by the pipeline. The `CRDS_SERVER_URL` variable specifies from which CRDS server reference files should be obtained. For more information, see *Reference Files, Parameter Files and CRDS*.

4. Running the Pipeline

With `jwst` installed and CRDS configured for JWST, you can now run the pipeline and use JWST `datamodels`.

For information on how to run the pipeline using the Python interface, see *Running the JWST pipeline: Python Interface*.

For information on how to run the pipeline using the command line interface, see *Running the JWST pipeline: Command Line Interface*.

For information on how to read and write data files with JWST `datamodels`, see *JWST datamodels*.

CHAPTER

ONE

INSTALLATION

Stable releases of the `jwst` package are registered at PyPI (https://pypi.org/project/jwst/). The development version of `jwst` is installable from the Github repository (https://github.com/spacetelescope/jwst).

`jwst` is also available as part of stenv (https://stenv.readthedocs.io/en/latest/) (Space Telescope Environment).

1.1 Detailed Installation Instructions

The `jwst` package can be installed into a virtualenv or conda environment via `pip`. We recommend that for each installation you start by creating a fresh environment that only has Python installed and then install the `jwst` package and its dependencies into that bare environment. If using conda environments, first make sure you have a recent version of Anaconda or Miniconda installed (https://docs.conda.io/en/latest/miniconda.html). If desired, you can create multiple environments to allow for switching between different versions of the `jwst` package (e.g. a released version versus the current development version).

In all cases, the installation is generally a 3-step process

1. Create a conda environment
2. Activate that environment
3. Install the desired version of the `jwst` package into that environment

Details are given below on how to do this for different types of installations, including tagged releases, DMS builds used in operations, and development versions. Remember that all conda operations must be done from within a bash/zsh shell.

Warning: Users on MacOS Mojave (10.14) should limit their environment python to 3.9 - there is a package dependency which currently fails to build on Mojave with python>=3.10.

1.1.1 Installing Latest Release

You can install the latest released version via `pip`. From a bash/zsh shell:

```
>> conda create -n <env_name> python
>> conda activate <env_name>
>> pip install jwst
```

1.1.2 Installing Previous Releases

You can also install a specific version (from `jwst 0.17.0` onward):

```
>> conda create -n <env_name> python
>> conda activate <env_name>
>> pip install jwst==1.3.3
```

Installing specific versions before `jwst 0.17.0` need to be installed from Github:

```
>> conda create -n <env_name> python
>> conda activate <env_name>
>> pip install git+https://github.com/spacetelescope/jwst@0.16.2
```

1.1.3 Installing the Development Version from Github

You can install the latest development version (not as well tested) from the Github master branch:

```
>> conda create -n <env_name> python
>> conda activate <env_name>
>> pip install git+https://github.com/spacetelescope/jwst
```

1.1.4 Upgrading Installed Version

Important: Do NOT use `pip install jwst --upgrade` to upgrade your installation. This does not check if dependencies are upgraded and will cause issues. Instead, use the method detailed below.

If you have previously installed `jwst` and you would like to upgrade to keep your install up-to-date, we recommend that you first uninstall the package in your environment of choice and then reinstall:

```
>> pip uninstall jwst
>> pip install jwst
```

This will ensure that all dependency packages are also upgraded. This also applies when using the development version of jwst - to upgrade and grab recent changes, uninstall and re-install the master branch from Github:

```
>> pip uninstall jwst
>> pip install git+https://github.com/spacetelescope/jwst
```

1.1.5 Installing with `stenv`

`jwst` is available as part of `stenv`, a set of installable Conda environments that bundle software for astronomical data analysis with JWST, HST, and other observatories. See the stenv documentation (https://stenv.readthedocs.io/en/latest/) for more information.

For more install instructions, including how to install jwst for development or how to install a DMS operational build, see the Github README (https://github.com/spacetelescope/jwst).

CHAPTER

TWO

CONTRIBUTING AND REPORTING ISSUES

`jwst` is open source - if you would like to contribute code or file an issue, please see the the Github Contribution Guide (https://github.com/spacetelescope/jwst/blob/master/CONTRIBUTING.md).

CHAPTER

THREE

INTRODUCTION TO THE JWST PIPELINE

3.1 Introduction

The JWST Science Calibration Pipeline processes data from all JWST instruments and observing modes by applying various science corrections sequentially, producing both fully-calibrated individual exposures and high-level data products (mosaics, extracted spectra, etc.). The pipeline is written in Python, hosted open-source on Github, and can be run either via *command line interface* (`strun`) or via the *Python interface*.

The full end-to-end 'pipeline' (from raw data to high-level data products) is comprised of three seperate pipeline stages that are run individually to produce output products at different calibration levels:

Stage 1
: Detector-level corrections and ramp fitting for individual exposures.

Stage 2
: Instrument-mode calibrations for individual exposures.

Stage 3
: Combining data from multiple exposures within an observation

As such, the term 'pipeline' may refer to a single pipeline stage or to the full three-stage series.

Because JWST has many different instruments and observing modes, there are several different pipeline modules available for each stage. There is one single pipeline for Stage 1 - corrections are applied nearly universally for all instruments and modes. There are two pipeline modules for Stage 2: one for imaging and one for spectroscopic modes. Stage 3 is divided into five separate modules for imaging, spectroscopic, coronagraphic, Aperture Masking Interferometry (AMI), and Time Series Observation (TSO) modes. Details of all the available pipeline modules can be found at *Pipeline Modules*.

Each pipeline stage consists of a series of sequential steps (e.g, saturation correction, ramp fitting). Each full pipeline stage and every individual step has a unique module name (i.e Detector1Pipeline, or DarkCurrentCorrection). Steps can also be run individually on data to apply a single correction. The output of each pipeline stage is the input to the next, and within a pipeline stage the output of each step is the input to the next.

The pipeline relies on three components to direct processing: input data, step parameters, and reference files. The inputs to the pipeline modules are individual exposures (`fits` files) or associations of multiple exposures (`asn.json` files). The parameters for each pipeline step are determined hierarchically from the parameter defaults, parameter reference files, and any specified overrides at run time. Finally, reference files provide data for each calibration step that is specific to the dataset being processed. These files may depend on things like instrument, observing mode, and date. In both the command line and Python interface, a pipeline or step module may be configured before running. Reference files can be overridden from those chosen by CRDS, steps in a pipeline can be skipped, step parameters can be changed, and the output and intermediate output products can be controlled.

A pipeline (or individual step) outputs corrected data either by writing an output file on disk or returning an in-memory datamodel object. The output file suffix (i.e `cal.fits`, `rate.fits`) depends on level of calibration - each full pipeline

stage as well as each individual step have a unique file suffix so that outputs may be obtained at any level of calibration. Other pipeline outputs include photometry catalogs and alignment catalogs (at stage 3).

3.2 Overview of Pipeline Code

The following is a brief overview of how the pipeline code in `jwst` is organized.

Pipeline and Step Classes

The JWST pipeline is organized into two main classes - `pipeline` classes and `step` classes. Pipelines are made up of sequential `step` classes chained together, the output of one step being piped to the next, but both pipelines and steps are represented as objects that can be configured and run on input data.

```
Detector1Pipeline  # an example of a pipeline class
DarkCurrentStep    # an example of a step class
```

Each pipeline or step has a unique module name, which is the identifier used to invoke the correct pipeline/step when using either the Python or the Command Line Interface.

Package Structure

Within the `jwst` repository, there are separate modules for each pipeline step. There is also a `pipeline` module, where the `pipeline` classes, consisting of `step` classes called in sequence, are defined.

```
jwst/
        assign_wcs/
                assign_wcs_step.py  # contains AssignWcsStep
                ...
        dark_current/
                dark_current_step.py  # contains DarkCurrent Step
                ...
        pipeline/
                calwebb_detector1.py  # contains Detector1Pipeline
                calwebb_image2.py  # contains Image2Pipeline
        ...
```

Dependencies

The `jwst` package has several dependencies (see the `pyproject.toml` file in the top-level directory of `jwst` for a full list). Some notable dependencies include:

asdf

ASDF (https://asdf.readthedocs.io/en/latest/), the Advanced Scientific Data Format is the file format the JWST uses to encode world coordinate system (WCS) information.

gwcs

GWCS (https://gwcs.readthedocs.io/en/latest/), Generalized World Coordinate System - is an generalized alternative to FITS WCS which makes use of astropy models to describle the translation between detector and sky coordinates. In JWST data, WCS information is encoded in an ASDF extension in the FITS file that contains GWCS object. In contrast, FITS WCS is limited because it stores the WCS transformation as header keywords, which is not sufficient to describe many of the transformations JWST uses.

stpipe

STPIPE (https://github.com/spacetelescope/stpipe) contains base classes for `pipeline` and `step`, and command line tools that are shared between the JWST and Nancy Grace Roman Telescope (https://roman-pipeline.readthedocs.io/en/latest/) (Roman) pipelines.

stcal

The `stcal` package contains step code that is common to both JWST and the Roman telescope, to avoid redundancy. All step classes for the JWST pipeline are still defined in `jwst`, but some of the underlying code for these steps lives in `stcal` if the algorithm is shared by Roman (for example, ramp fitting, saturation).

CHAPTER

FOUR

REFERENCE FILES, PARAMETER FILES AND CRDS

The JWST pipeline uses version-controlled *reference files* and *parameter files* to supply pipeline steps with necessary data and set pipeline/step parameters, respectivley. These files both use the ASDF format, and are managed by the Calibration References Data System (*CRDS*) system.

4.1 Reference Files

Most pipeline steps rely on the use of reference files that contain different types of calibration data or information necessary for processing the data. The reference files are instrument-specific and are periodically updated as the data processing evolves and the understanding of the instruments improves. They are created, tested, and validated by the JWST Instrument Teams. The teams ensure all the files are in the correct format and have all required header keywords. The files are then delivered to the Reference Data for Calibration and Tools (ReDCaT) Management Team. The result of this process is the files being ingested into the JWST Calibration Reference Data System (CRDS), and made available to users, the pipeline team and any other ground subsystem that needs access to them.

Information about all the reference files used by the Calibration Pipeline can be found at *Reference File Information*, as well as in the documentation for each Calibration Step that uses a reference file. Information on reference file types and their correspondence to calibration steps is described within the table at *Reference File Types*.

4.2 Parameter Files

Parameter files, which like reference files are encoded in ASDF and version-controlled by CRDS, define the 'best' set of parameters for pipeline steps as determined by the JWST instrument teams, based on insturment, observing model, filter, etc. They also may evolve over time as understanding of caibration improves.

By default, when running the pipeline via `strun` or using the `pipeline/step.call()` method when using the Python interface, the appropriate parameter file will be determined and retrieved by CRDS to set step parameters.

4.3 CRDS

Calibration References Data System (CRDS) is the system that manages the reference files that the pipeline uses. For the JWST pipeline, CRDS manages both data reference files as well as parameter reference files which contain step parameters.

CRDS consists of external servers that hold all available reference files, and the machinery to map the correct reference files to datasets and download them to a local cache directory.

When the Pipeline is run, CRDS uses the metadata in the input file to determine the correct reference files to use for that dataset, and downloads them to a local cache directory if they haven't already been downloaded so they're available on your filesystem for the pipeline to use.

The environment variables `crds_context` and `crds_server` must be set before running the pipeline

4.3.1 Reference Files Mappings (CRDS Context)

One of the main functions of CRDS is to associate a dataset with its best reference files - this mapping is referred to as the 'CRDS context' and is defined in a `pmap` file, which itself is version-controlled to allow access to the reference file mapping at any point in time, and revert to any previous set of reference files if desired.

The CRDS context is usually set by default to always give access to the most recent reference file deliveries and selection rules - i.e the 'best', most up-to-date set of reference files. On occasion it might be necessary or desirable to use one of the non-default mappings in order to, for example, run different versions of the pipeline software or use older versions of the reference files. This can be accomplished by setting the environment variable `CRDS_CONTEXT` to the desired project mapping version, e.g.

```
$ export CRDS_CONTEXT='jwst_0421.pmap'
```

For all information about CRDS, including context lists, see the JWST CRDS website:

https://jwst-crds.stsci.edu/

4.3.2 CRDS Servers

The CRDS server can be found at

```
https://jwst-crds.stsci.edu
```

Inside the STScI network, the pipeline defaults are sufficient and no further action is necessary.

To run the pipeline outside the STScI network, CRDS must be configured by setting two environment variables:

- CRDS_PATH: Local folder where CRDS content will be cached.
- CRDS_SERVER_URL: The server from which to pull reference information

To setup to use the server, use the following settings:

```
export CRDS_PATH=$HOME/crds_cache/
export CRDS_SERVER_URL=https://jwst-crds.stsci.edu
```

4.3.3 Setting CRDS Environment Variables in Python

The CRDS environment variables need to be defined *before* importing anything from `jwst` or `crds`. The examples above show how to set an environment variable in the shell, but this can also be done within a Python session by using `os.envron`. In general, any scripts should assume the environment variables have been set before the scripts have run. If one needs to define the CRDS environment variables within a script, the following code snippet is the suggested method. These lines should be the first executable lines:

```
import os
os.environ['CRDS_PATH'] = 'path_to_local_cache'
os.environ['CRDS_SERVER_URL'] = 'url-of-server-to-use'
```

(continues on next page)

(continued from previous page)

```
# Now import anything else needed
import jwst
```

CHAPTER

FIVE

PARAMETERS

Parameters, which exist at both the step level and the global pipeline level, can be set to change various aspects of processing. Parameters can be set in a parameter file, on the command line, or passed in as an argument when running in Python. Note that because there are multiple ways to set parameters, there is a hierarchy involved - overrides set on a pipeline or step object will take precedence over values in a parameter file. See *Parameter Precedence* for a full description of how a parameter gets its final value.

If there is need to re-use a set of parameters often, parameters can be stored in **parameter files**. See *Parameter Files* for more information.

To see what parameters are available for any given pipeline or step, use the `-h` option on `strun`. Some examples are:

```
$ strun calwebb_detector1 -h
$ strun jwst.dq_init.DQInitStep -h
```

5.1 Universal Parameters

The set of parameters that are common to all pipelines and steps are referred to as **universal parameters** and are described below. When these parameters are set at the pipeline level, they will apply to all steps within that pipeline, unless explicitly overridden for a specific step.

5.1.1 Output Directory

By default, all pipeline and step outputs will drop into the current working directory, i.e., the directory in which the process is running. To change this, use the `output_dir` parameter. See .. _python_output_directory: for instructions when running in Python, and .. _cli_output_directory: for instructions using the command line interface.

5.1.2 Output File

When running a pipeline, the `stpipe` infrastructure automatically passes the output data model from one step to the input of the next step, without saving any intermediate results to disk. If you want to save the results from individual steps, you have two options:

1. Specify `save_results`. This option will save the results of the step, using a filename created by the step.
2. Specify a file name using `output_file <basename>`. This option will save the step results using the name specified.

To do this using the Python pipeline interface, see .. _python_output_file:. To do this when using the command line interface, see .. _cli_output_file:.

5.1.3 Override Reference File

For any step that uses a calibration reference file you always have the option to override the automatic selection of a reference file from CRDS and specify your own file to use. Parameters for this are of the form `--override_<ref_type>`, where `ref_type` is the name of the reference file type, such as `mask`, `dark`, `gain`, or `linearity`. When in doubt as to the correct name, just use the `-h` argument to `strun` to show you the list of available override parameters.

To override the use of the default linearity file selection, for example, you would use:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
       --steps.linearity.override_linearity='my_lin.fits'
```

5.1.4 Skip

Another parameter available to all steps in a pipeline is `skip`. If `skip=True` is set for any step, that step will be skipped, with the output of the previous step being automatically passed directly to the input of the step following the one that was skipped. For example, if you want to skip the linearity correction step, one can specify the `skip` parameter for the `strun` command:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --steps.linearity.skip=True
```

Alternatively, if using a *parameter file*, edit the file to add the following snippet:

```
steps:
- class: jwst.linearity.linearity_step.LinearityStep
  parameters:
    skip: true
```

5.2 Pipeline/Step Parameters

All pipelines and steps have **parameters** that can be set to change various aspects of how they execute. To see what parameters are available for any given pipeline or step, use the `-h` option on `strun`. Some examples are:

```
$ strun calwebb_detector1 -h
$ strun jwst.dq_init.DQInitStep -h
```

To set a parameter, simply specify it on the command line. For example, to have *calwebb_detector1* save the calibrated ramp files, the `strun` command would be as follows:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits --save_calibrated_
↪ramp=true
```

To specify parameter values for an individual step when running a pipeline use the syntax `--steps.<step_name>.<parameter>=value`. For example, to override the default selection of a dark current reference file from CRDS when running a pipeline:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
      --steps.dark_current.override_dark='my_dark.fits'
```

If there is need to re-use a set of parameters often, parameters can be stored in **parameter files**. See *Parameter Files* for more information.

5.3 Pipeline/Step Parameters

CHAPTER

SIX

RUNNING THE JWST PIPELINE: PYTHON INTERFACE

Note: The use of the `run` method to run a pipeline or step is not recommended. By default, using the `pipeline.run()` method defaults to pipeline and step-level coded defaults, ignoring parameter files, unless explicitly overridden. Please see *Advanced use - pipeline.run() vs. pipeline.call* for more details.

The Python interface is one of two options for running the pipeline. See *here* for an overview of the alternative command line interface.

6.1 Overview of Running the Pipeline in Python

When using the Python interface to the JWST pipeline, each `pipeline` and `step` is available as a module that can be imported into your Python session, configured (either directly as arguments/attributes or with a *parameter file*), and used to process input data. The following section will describe the necessary steps to run a pipeline or step in Python.

6.1.1 CRDS Environment Variables

The CRDS environment variables need to be defined *before* importing anything from `jwst` or `crds` to allow access to reference and parameter files. These environment variables can be set in the shell, or in a Python session by using `os.environ` (https://docs.python.org/3/library/os.html#os.environ). See *Setting CRDS Environment Variables in Python* for more information.

6.1.2 Importing and Running Pipelines and Steps in Python

All full pipeline stages can be imported by name from the `pipeline` module:

```
from jwst.pipeline import Image3Pipeline
from jwst.pipeline import Spec2Pipeline
```

Individual pipeline steps can be imported by name from their respective module in `jwst`:

```
from jwst.saturation import SaturationStep
from jwst.ramp_fitting import RampFitStep
```

Details of all the available pipeline modules and their names can be found at *Pipeline Modules*.

Once *imported*, you can execute a pipeline or a step from within Python by using the .call() method of the class. The input can be either a string path to a file on disk or an open `DataModel` object. Note that the .run() class method is also available for use, but is discouraged and should be used only with caution (see *here* for more information).

Example: Running a Pipeline or Step with Default Parameters and Reference Files

```
# running a full pipeline stage, input is path to file
from jwst.pipeline import Detector1Pipeline
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits')

# running a single pipeline step, input is datamodel object
from jwst.linearity import LinearityStep
import stdatamodels.jwst.datamodels as dm
input_model = dm.open('jw00001001001_01101_00001_mirimage_uncal.fits')
result = LinearityStep.call(input_model)
```

In the examples above, the returned value `result`, is a `Datamodel` containing the corrected data - no files are written out, by default. See *Controlling Output File Behavior* for information on how to control the generation of output files.

Additionally in both examples above, there are no arguments other than the input data being passed in to the `call` method, so the appropriate parameter files and reference files are chosen by CRDS based on the *current context*. The *following section* will show how to configure the pipeline to override these defaults.

6.2 Configuring a Pipeline/Step in Python

By default when using the `.call()` method to run a pipeline/step, pipeline/step parameters and reference files are chosen by CRDS based on instrument, observing mode, date, etc. If set to the most current *context*, these represent the 'best' set of parameters and reference files for the dataset passed in, as determined by the JWST instrument teams.

To override parameter and reference file defaults, a pipeline/step can be configured for custom processing. Pipeline-level and step-level parameters can be changed, output file behavior can be set, reference files can be overridden, and pipeline steps can be skipped if desired. This section will be a general overview on how to configure the pipeline when running in Python, and the following sections will elaborate on each of these options.

When running in Python, there are two ways two configure a Pipeline/Step.

1. By passing in keyword arguments to a pipeline/step's `call` method
2. By using a *parameter file*

A combination of both keyword arguments and custom parameter files can be used for configuration, but keep in mind the hierarchy of *parameter precedence* to keep track of which value will get used if set in multiple locations.

Example: Configuring a pipeline/step with keyword arguments

```
# configuring a pipeline and the steps within the pipeline with keyword arguments
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                save_results=False,
                                steps={'jump': {'rejection_threshold': 12.0, 'save_
↪results':True}})
# configuring a pipeline step with keyword arguments
result = JumpStep.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                       save_results=True, 'rejection_threshold'=12.0)
```

Both examples above show how to configure the jump detection step with the same settings - the `rejection_threshold` set to 12.0, and `save_results` set to True to indicate the result from the step should be written to an output file.

The first example shows when the jump step is run inside a pipeline - because a pipeline consists of many steps, parameters for a substep are specified within the `steps` argument, a nested dictionary keyed by each substep and again by each possible parameter for each substep. Pipeline-level arguments (in this case, `save_results`) are passed in

individually as keyword arguments. Note that in this example, the 'save_results' argument within `steps` will override the pipeline-level 'save_results' argument.

The second example shows the same configuration to the jump step, but this time when the step is run standalone. Here, there is no `steps` dictionary argument and all arguments can be passed to the step directly since it is now at the step level.

Example: Configuring a pipeline/step with a parameter file

To use a custom parameter file, set the `config_file` parameter:

```
# passing a custom parameter file to a pipeline
result = Detector1Pipeline.call("jw00017001001_01101_00001_nrca1_uncal.fits",\
                                config_file='calwebb_detector1.asdf')
```

Again, note the *parameter precedence* rules. If an override parameter file passed in does not contain the full set of required parameters for a step, the others will be obtained according to those rules and may grab values from the CRDS-chosen parameter file as well. If a custom parameter file is passed in to `config_file` AND an argument is passed directly to the pipeline/step class, the value in the parameter file is overridden.

6.2.1 Setting Step Parameters on a Pipeline or Individual Step

All steps have parameters that can be set to change various aspects of how they execute (e.g switching on and off certain options in a step, setting thresholds). By default, the values of these parameters are set in the CRDS-chosen parameter file (and if absent, defer to the coded defaults), but they can be overridden if desired.

As Arguments to a Pipeline / Step

As discussed in *above*, when setting a step-level parameter when that step is a substep of a pipeline, it must be passed to the `steps` argument dictionary. For exaple, to change the `rejection_threshold` parameter of the jump detection step when running the full Detector1Pipeline:

```
from jwst.pipeline import Detector1Pipeline
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                steps={'jump': {'rejection_threshold':12.0)}})
```

When running a single step, step-level parameters can be passed in directly as keyword arguments. For example, to change the parameter `rejection_threshold` for the jump detection step when running the step individually:

```
from jwst.jump import JumpStep
result = JumpStep.call('jw00017001001_01101_00001_nrca1_uncal.fits', rejection_
↪threshold=12.0)
```

Using a Parameter File

Alternatively, if using a *parameter file*, edit the file to add the following snippet (in this example, to a file named `my_config_file.asdf` in the current working directory):

```
steps:
- class: jwst.jump.jump_step.JumpStep
  parameters:
    rejection_threshold : 12
```

And pass in the modified file to the `config_file` argument:

```
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                config_file='my_config_file.asdf')
```

Disabling all CRDS Step Parameters

Retrieval of Step parameters from CRDS can be completely disabled by setting the STPIPE_DISABLE_CRDS_STEPPARS environment variable to TRUE. This can be done in the shell, or using the os.environ() command:

```
os.environ["STPIPE_DISABLE_CRDS_STEPPARS"] = 'True'
```

6.2.2 Overriding Reference Files

To override the reference file for a step selected by CRDS:

As Arguments to a Pipeline / Step

To override a reference file for a step within a pipeline, for example the `saturation` step in the Detector1Pipeline the `override_saturation` argument can be set in the `saturation` section of the `steps` argument.

```
# To override a reference file of a step within a pipeline
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                 steps={"saturation" : {"override_saturation": '/path/to/
↪new_saturation_ref_file.fits'}})
```

Multiple reference file overrides can be provided, for example:

```
# To override a reference file for multiple steps within a pipeline
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                                                steps={"saturation": {
↪"override_saturation": '/path/to/new_saturation_ref_file.fits'},
                                                                       {"jump" : {
↪"override_jump": '/path/to/new_jump_ref_file.fits'}})
```

To override a reference file for a standalone step, "override_<stepname>" can be passed directly as a keyword argument to that step's `call` method:

```
# To override a reference file when running a standalone step
 from jwst.linearity import SaturationStep
 SaturationStep.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                          override_saturation='/path/to/new_saturation_
↪ref_file.fits')
```

Using a Parameter File

If using a *parameter file* for configuration, to override a reference edit the file to add the following snippet (in this example, to a file named `my_config_file.asdf` in the current working directory):

```
steps:
- class: jwst.linearity.saturation_step.SaturationStep
  parameters:
    override_saturation: '/path/to/new_saturation_ref_file.fits'
```

And pass in the modified file to the `config_file` argument:

```
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                config_file='my_config_file.asdf')
```

To use an entire set of past reference files from a previous CRDS mapping, see *here*.

6.2.3 Skipping a Pipeline Step

Note: Some steps in a pipeline expect certain previous steps to have been run beforehand, and therefore won't run if that expected previous correction has not been applied. Proceed with caution when skipping steps.

When using the Python interface you wish to run a pipeline but skip one or some of the steps contained in that pipeline, this can be done in two different ways:

As Arguments to a Pipeline / Step

Every step in a pipeline has a `skip` parameter that when set to true, will entirely skip that step. For example, to skip the saturation step in the Detector1Pipeline:

```
# To set a step parameter on a step within a pipeline
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits', steps={
↪"saturation": {"skip": True}})
```

Using a Parameter File

The equivalent to the above example can be done by adding the following snippet to your parameter file (in this example, to a file named `my_config_file.asdf` in the current working directory):

```
steps:
- class: jwst.linearity.linearity_step.LinearityStep
  parameters:
    skip: true
```

And pass in the modified file to the `config_file` argument:

```
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                config_file='my_config_file.asdf')
```

6.3 Controlling Output File Behavior

By default, when running in Python, all outputs are returned in-memory (typically as a `Datamodel`) and no output files are written - even the final result of a pipeline. To control this behavior, and other aspects of output file generation like directory and file name, certain pipeline and step-level parameters can be set.

Output file behavior can be modified with the ``save_results``, ``output_file``, and ``output_dir`` parameters

6.3.1 Saving Final Pipeline Results

The `save_results` parameter, when set at the pipeline-level, indicates that the final pipeline output products should be saved to a file. The output files will be in the current working directory, and be named based on the input file name and the appropriate file suffix. Note that setting `save_results` at the pipeline-level will not save the results from each step, only the final results from the full pipeline.

```
# To save the final results from a pipeline to a file
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits', save_
↪results=True)
```

In this example, the following output files will be written in the current working directory:

- `jw00017001001_01101_00001_nrca1_trapsfilled.fits`
- `jw00017001001_01101_00001_nrca1_rate.fits`
- `jw00017001001_01101_00001_nrca1_rateints.fits`

Changing Output File Name

Setting `output_file` at the pipeline-level indicates that the pipeline's final result should be saved (so, also setting `save_results` is redundant), and that a new file base name should be used with the appropriate file suffix appended. For example, to save the intermediate result from the saturation step when running `Detector1Pipeline` with a file name based on the string `detector_1_final` instead of `jw00017001001_01101_00001_nrca1`:

```
# saving the final results from running a pipeline with a custom output file basename
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits', output_
↪file='detector_1_final_result')
```

In this example, the following output files will be written in the current working directory

- `detector_1_final_result_trapsfilled.fits`
- `detector_1_final_result_rate.fits`
- `detector_1_final_result_rateints.fits`

Changing Output File Directory

When set at the pipeline level, the `output_dir` parameter will set where the final pipeline output products are placed. The default is the current working directory. For example, to save the results from Detector1Pipeline in a subdirectoy `/calibrated`:

Setting `output_dir` at the pipeline-level indicates that the pipeline's final results should be saved (so, also setting `save_results` is redundant), and that the files should be saved in the directory specified instead of the current working directory. For example, to save the intermediate results of `Detector1Pipeline` in a subdirectory `/calibrated`:

```
# to save the final result of a pipeline in a different specified output directory
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits', output_
↪dir='calibrated')
```

6.3.2 Saving Intermediate Step Results

When the `save_results` parameter is set at the step-level (either within a pipeline, or on a standalone step), it indicates that the result from that step should be saved to a file.

To save the intermediate output from a step within a pipeline:

```
# To save the intermediate results of a step within a pipeline to a file
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                                                steps={"saturation": {
↪"save_results": True}})
```

Similarly, when `save_results` is set on an individual step class, this will indicate that the final result from that step should be saved.

```
# To save the final results from SaturationStep when run standalone
 from jwst.linearity import SaturationStep
 SaturationStep.call('jw00017001001_01101_00001_nrca1_uncal.fits', save_results=True)
```

Setting Output File Name

Setting `output_file` at the step-level indicates that the step's result should be saved (so, also setting `save_results` is redundant), and that a new file base name should be used with the appropriate file suffix appended. For example, to save the intermediate result from the saturation step when running `Detector1Pipeline` with a file name based on the string `saturation_result` instead of `jw00017001001_01101_00001_nrca1`:

```
# To save the intermediate results of a step within a pipeline to a file with a custom␣
↪name
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                                                steps={"saturation": {
↪"output_file": 'saturation_result'})
```

Similarly, when `output_file` is set on an individual step class, this will indicate that the result from that step should be saved to a file with that basename and the appropriate suffix.

```
# To save the final results from SaturationStep with a custom output file name when run␣
↪standalone
 from jwst.linearity import SaturationStep
 SaturationStep.call('jw00017001001_01101_00001_nrca1_uncal.fits', output_file=
↪"saturation_result")
```

Setting Output File Directory

Setting `output_dir` at the step-level indicates that the step's result should be saved (so, also setting `save_results` is redundant), and that the files should be saved in the directory specified instead of the current working directory. For example, to save the intermediate results of `DarkCurrentStep` when running `Detector1Pipeline` in a subdirectory `/calibrated`:

```
# to save the intermediate step result in a different specified output directory
 from jwst.pipeline import Detector1Pipeline
 result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                                                  steps={'dark': {'output_
↪dir': 'calibrated'}})
```

Similarly, when `output_dir` is set on an individual step class, this will indicate that the result from that step should be saved to the specified directory:

```
# to save the final result of a
 from jwst.pipeline import Detector1Pipeline
 result = DarkCurrentStep.call('jw00017001001_01101_00001_nrca1_uncal.fits', output_dir=
↪'calibrated')
```

6.4 Advanced use - `pipeline.run()` vs. `pipeline.call`

Another option for running pipelines or steps is to use the `run()` method instead of the `call()` method. **Using .run() is not reccomended** and considered advanced use, but it is an option to users.

The difference between `.run()` in `.call()` is in the retrieval and use of parameters from CRDS parameter files. When the `.call()` method is invoked, there is additional setup done to retrieve parameter and reference files and reconcile those with any passed into the pipeline directly as an argument or in a custom parameter file. When `.call()` is invoked, a new instance of the pipeline/step class is created internally, and after parameters are determined the `.run()` method of that internal class is called. Because the actual processing occurs on this new instance, attributes cannot be set directly on the original pipeline/step class. They must be passed in as arguments to `.call()` or set in the parameter file.

In contrast, when using the `.run()` method directly on a pipeline/step, the additional logic to determine parameters and reference files is skipped. The pipeline instance is being run as-is, and coded defaults for the pipeline and each intermediate step will be used unless explicitly overridden individually. Because the instance created is being run directly on the data, attributes can be set directly:

```
from jwst.pipeline import Detector1Pipeline
pipe = Detector1Pipeline()
pipe.jump.rejection_threshold = 12
pipe.ramp_fit.skip = True
result = pipe.run('jw00017001001_01101_00001_nrca1_uncal.fits')
```

The `pipe` object created and the attributes set will persist and this object can be reused within a Python session for processing data. Keep in mind that each individual step parameter must be set when using this method, or else the coded defaults will be used, which may be inappropriate for the dataset being processed.

See *Executing a pipeline or pipeline step via call()* for more information.

CHAPTER

SEVEN

RUNNING THE JWST PIPELINE: COMMAND LINE INTERFACE (STRUN)

Note: For seasoned users who are familiar with using `collect_pipeline_cfgs` and running pipelines by the default configuration (CFG) files, please note that this functionality has been deprecated. Please read *CFG Usage Deprecation Notice*.

Individual steps and pipelines (consisting of a series of steps) can be run and configured from the command line using the `strun` command. `strun` is one of two options for running the pipeline. See *here* for an overview of the alternative Python interface.

7.1 CRDS Environment Variables

The CRDS environment variables need to be defined *before* running a pipeline or step with `strun` to allow the pipeline to access reference and parameter files. See *CRDS* for more information.

7.2 Overview of Running the Pipeline with `strun`

The first argument to `strun` must be one of either a pipeline name, Python class of the step or pipeline to be run, or the name of a parameter file for the desired step or pipeline (see *Parameter Files*). The second argument to `strun` is the name of the input data file to be processed.

```
$ strun <pipeline_name, class_name, or parameter_file> <input_file>
```

Pipeline classes also have a **pipeline name**, or **alias**, that can be used instead of the full class specification. For example, `jwst.pipeline.Detector1Pipeline` has the alias `calwebb_detector1` and can be run as

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
```

A full list of pipeline aliases can be found in *Pipeline Stages*.

7.3 Exit Status

`strun` produces the following exit status codes:

- 0: Successful completion of the step/pipeline
- 1: General error occurred
- 64: No science data found

The "No science data found" condition is returned by the `assign_wcs` step of the `calwebb_spec2` pipeline when, after successfully determining the WCS solution for a file, the WCS indicates that no science data will be found. This condition most often occurs with NIRSpec's Multi-object Spectroscopy (MOS) mode: There are certain optical and MSA configurations in which dispersion will not cross one or the other of NIRSpec's detectors.

7.3.1 Configuring a Pipeline/Step with `strun`

By default, pipeline parameters and reference files are chosen by CRDS based on instrument, observing mode, date, etc. If set to the most current *Reference Files Mappings (CRDS Context)*, these represent the 'best' set of parameters and reference files for the pipeline as determined by the JWST instrument teams.

A Pipeline/Step can be configured for custom processing. Pipeline-level and step-level parameters can be changed, output file behavior can be set, references files can be overridden, and pipeline steps can be skipped if desired. This section will be a general overview on how to configure the pipeline when running with `strun`, and the following sections will elaborate on each of these possible customizations and demonstrate usage.

When running command line with ``strun``, there are two ways two configure a Pipeline/Step.

1. By passing in arguments to a pipeline/step on the command line
2. By using a *parameter file* and passing this in as an argument on the command line

A combination of arguments and custom parameter files can be used for configuration, but keep in mind the hierarchy of *parameter precedence* to keep track of which value will get used if set in multiple locations.

7.4 Setting Step Parameters on a Pipeline or Individual Step

All pipelines and steps have parameters that can be set to change various aspects of how they execute (e.g switching on and off certain options in a step, setting thresholds). By default, the values of these parameters are set in the CRDS-chosen parameter file, but they can be overridden when running the pipeline with `strun`. As mentioned, this can either be done by passing in command line arguments or by passing in a custom parameter file - both methods will be described in this section.

Using Command Line Arguments

When running a pipeline, step-level parameters can be changed by passing in a command line argument to that step. For example, to change the `rejection_threshold` parameter of the jump detection step when running the full Detector1Pipeline:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --steps.jump.rejection_threshold=12.0
```

When running a standalone step, command line arguments do not need to be nested within `steps`. For example, to change the parameter `rejection_threshold` for the jump detection step when running the step individually:

```
$ strun jump jw00017001001_01101_00001_nrca1_uncal.fits --rejection_threshold=12.0
```

Using a Parameter File

Alternatively, if using a *parameter file*, edit the file to add the following snippet (in this example, to a file named 'my_config_file.asdf' in the current working directory):

```
steps:
- class: jwst.jump.jump_step.JumpStep
  name: jump
  parameters:
    rejection_threshold : 12
```

And pass in the modified file to `strun`:

```
$ strun my_config_file.asdf jw00017001001_01101_00001_nrca1_uncal.fits
```

7.5 Overriding Reference Files

By default, when the pipeline or step is run, CRDS will determine the best set of reference files based on file metadata and the current CRDS mapping (also known as 'context'). It is possible to override these files and use a custom reference file, or one not chosen by CRDS.

Using Command Line Arguments

For any step that uses a calibration reference file you always have the option to override the automatic selection of a reference file from CRDS and specify your own file to use. Parameters for this are of the form `--override_<ref_type>`, where `ref_type` is the name of the reference file type, such as `mask`, `dark`, `gain`, or `linearity`. When in doubt as to the correct name, just use the `-h` argument to `strun` to show you the list of available override parameters.

To override the use of the default linearity reference file selection with a custom file in the current working directory called `my_lin.fits`, for example, you would do:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
       --steps.linearity.override_linearity='my_lin.fits'
```

Or, if running the step individually, to override the reference file:

```
$ strun linearity jw00017001001_01101_00001_nrca1_uncal.fits
       --override_linearity='my_lin.fits'
```

Using a Parameter File

If using a *parameter file* for configuration, to override a reference edit the file to add the following snippet (in this example, to a file named 'my_config_file.asdf' in the current working directory):

```
steps:
- class: jwst.saturation.saturation_step.SaturationStep
  name: saturation
  parameters:
    override_saturation: '/path/to/new_saturation_ref_file.fits'
```

And pass in the modified file to `strun`:

```
$ strun my_config_file.asdf jw00017001001_01101_00001_nrca1_uncal.fits
```

To use an entire set of past reference files from a previous CRDS mapping, see *here*.

7.6 Skipping a Pipeline Step

Note: Some steps in a pipeline expect certain previous steps to have been run beforehand, and therefore won't run if that expected previous correction has not been applied. Proceed with caution when skipping steps.

When running a pipeline with `strun`, one or several steps within that pipeline can be skipped.

Using Command Line Arguments

Every step in a pipeline has a `skip` parameter that when set to true, will entirely skip that step. For example, to skip the saturation step in the Detector1Pipeline:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --steps.saturation.skip=True
```

Using a Parameter File

The equivalent to the above example can be done by adding the following snippet to your parameter file (in this example, to a file named 'my_config_file.asdf' in the current working directory):

```
steps:
- class: jwst.saturation.saturation_step.SaturationStep
  parameters:
    skip: true
```

And pass in the modified file to the `config_file` argument:

```
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                                config_file='my_config_file.asdf')
```

7.6.1 Controlling Output File Behavior with `strun`

By default, when running the pipeline with `strun`, the final outputs of a pipeline (or final outputs when running an individual step) will be written out to a file in the current working directory. The base name of these final output files is derived from the input file name, by default. Additionally, no intermediate step results will be saved. This behavior can be modified to change output file names, locations, and specify that intermediate results from a step in a pipeline should be written out to a file.

7.7 Saving Intermediate Pipeline Results to a File

The `stpipe` infrastructure automatically passes the output data model from one step to the input of the next step, without saving any intermediate results to disk. If you want to save the results from individual steps, you have two options:

- Specify `save_results` on an individual step within the pipeline. This option will save the results of the step, using a filename created by the step.
- Specify a file name using `output_file` <basename> for an individual step. This option indicated that results should be saved, and to use the name specified.

For example, to save the result from the dark current step of `Detector1Pipeline` (using the *alias* name `calwebb_detector1`):

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --steps.dark_current.save_results=true
```

This will create the file `jw00017001001_01101_00001_dark_current.fits` in the current working directory.

7.8 Setting Output File Name

As demonstrated in the *section above*, the `output_file` parameter is used to specify the desired name for output files. When done at the step-level as shown in those examples, the intermediate output files from steps within a pipeline are saved with the specified name.

You can also specify a particular file name for saving the end result of the entire pipeline using the `--output_file` parameter:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --output_file='stage1_processed'
```

In this situation, using the default configuration, three files are created:

- `stage1_processed_trapsfilled.fits`
- `stage1_processed_rate.fits`
- `stage1_processed_rateints.fits`

When running a standalone step, setting `--output_file` at the top-level will determine the name of the final output product for that step, overriding the default based on input name:

```
$ strun linearity jw00017001001_01101_00001_nrca1_uncal.fits
    --output_file='intermediate_linearity'
```

Similarly, to save the result from a step within a pipeline (for example, the dark current step of `calwebb_detector1`) with a different file name:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --steps.dark_current.output_file='intermediate_result'
```

A file, `intermediate_result_dark_current.fits`, will then be created. Note that the name of the step will be appended as the file name suffix

7.9 Setting Output File Directory

To change the output directory of the final pipeline products from the default of the current working directory, use the `output_dir` option.

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --steps.dark_current.output_dir='calibrated'
```

When this is run, all three final output products of `Detector1Pipeline` will be saved within the subdirectory `calibrated`.

Setting `output_dir` at the step-level indicates that the step's result should be saved (so, also setting `save_results` is redundant), and that the files should be saved in the directory specified instead of the current working directory. For

example, to save the intermediate results of `DarkCurrentStep` when running `Detector1Pipeline` in a subdirectory `/calibrated`:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --steps.dark_current.output_dir='calibrated'
```

Similarly, when `output_dir` is set on an individual step class, this will indicate that the result from that step should be saved to the specified directory:

```
$ strun dark_current jw00017001001_01101_00001_nrca1_uncal.fits --output_dir='calibrated'
```

CHAPTER

EIGHT

AVAILABLE PIPELINES

There are many pre-defined pipeline modules for processing data from different instrument observing modes through each of the 3 stages of calibration. For all of the details see *Pipeline Stages*.

8.1 Pipeline/Step Suffix Definitions

However the output file name is determined (*see above*), the various stage 1, 2, and 3 pipeline modules will use that file name, along with a set of predetermined suffixes, to compose output file names. The output file name suffix will always replace any known suffix of the input file name. Each pipeline module uses the appropriate suffix for the product(s) it is creating. The list of suffixes is shown in the following table. Replacement occurs only if the suffix is one known to the calibration code. Otherwise, the new suffix will simply be appended to the basename of the file.

Product	Suffix
Uncalibrated raw input	uncal
Corrected ramp data	ramp
Corrected countrate image	rate
Corrected countrate per integration	rateints
Optional fitting results from ramp_fit step	fitopt
Background-subtracted image	bsub
Per integration background-subtracted image	bsubints
Calibrated image	cal
Calibrated per integration images	calints
CR-flagged image	crf
CR-flagged per integration images	crfints
Resampled 2D image	i2d
Resampled 2D spectrum	s2d
Resampled 3D IFU cube	s3d
1D extracted spectrum	x1d
1D extracted spectra per integration	x1dints
1D combined spectrum	c1d
Source catalog	cat
Segmentation map	segm
Time Series imaging photometry	phot
Time Series spectroscopic photometry	whtlt
Coronagraphic PSF image stack	psfstack
Coronagraphic PSF-aligned images	psfalign
Coronagraphic PSF-subtracted images	psfsub
AMI fringe and closure phases	ami
AMI averaged fringe and closure phases	amiavg
AMI normalized fringe and closure phases	aminorm

8.2 For More Information

More information on logging and running pipelines can be found in the `stpipe` User's Guide at *For Users*.

More detailed information on writing pipelines can be found in the `stpipe` Developer's Guide at *For Developers*.

If you have questions or concerns regarding the software, please open an issue at https://github.com/spacetelescope/jwst/issues or contact the JWST Help Desk (https://jwsthelp.stsci.edu).

CHAPTER

NINE

INPUT AND OUTPUT FILE CONVENTIONS

9.1 Input Files

There are two general types of input to any step or pipeline: references files and data files. The references files, unless explicitly overridden, are provided through CRDS.

Data files are the science input, such as exposure FITS files and association files. All files are assumed to be co-resident in the directory where the primary input file is located. This is particularly important for associations: JWST associations contain file names only. All files referred to by an association are expected to be located in the directory in which the association file is located.

9.2 Output Files

Output files will be created either in the current working directory, or where specified by the *output_dir* parameter.

File names for the outputs from pipelines and steps come from three different sources:

- The name of the input file
- The product name defined in an association
- As specified by the *output_file* parameter

Regardless of the source, each pipeline/step uses the name as a base name, onto which several different suffixes are appended, which indicate the type of data in that particular file. A list of the main suffixes can be *found below*.

The pipelines do not file manage versions. When re-running a pipeline, previous files will be overwritten.

9.2.1 Output Files and Associations

Stage 2 pipelines can take an individual file or an *association* as input. Nearly all Stage 3 pipelines require an association as input. Normally, the output file is defined in each association's "product name", which defines the basename that will be used for output file naming.

Often, one may reprocess the same set of data multiple times, such as to change reference files or parameters. When doing so, it is highly suggested to use `output_dir` to place the results in a different directory instead of using `output_file` to rename the output files. Most pipelines and steps create sets of output files. Separating runs by directory may be much easier to manage.

9.2.2 Individual Step Outputs

If individual steps are executed without an output file name specified via the `output_file` parameter, the `stpipe` infrastructure automatically uses the input file name as the root of the output file name and appends the name of the step as an additional suffix to the input file name. If the input file name already has a known suffix, that suffix will be replaced. For example:

```
$ strun jwst.dq_init.DQInitStep jw00017001001_01101_00001_nrca1_uncal.fits
```

produces an output file named `jw00017001001_01101_00001_nrca1_dq_init.fits`.

See *Pipeline/Step Suffix Definitions* for a list of the more common suffixes used.

CHAPTER

TEN

LOGGING CONFIGURATION

The name of a file in which to save log information, as well as the desired level of logging messages, can be specified in an optional configuration file. Two options exist - if the configuration file should be used for all instances of the pipeline, the configuration file should be named "stpipe-log.cfg". This file must be in the same directory in which you run the pipeline in order for it to be used.

If instead the configuration should be active only when specified, you should name it something other than "stpipe-log.cfg"; this filename should be specified using either the `--logcfg` parameter to the command line `strun` or using the `logcfg` keyword to a .call() execution of either a Step or Pipeline instance.

If this file does not exist, the default logging mechanism is STDOUT, with a level of INFO. An example of the contents of the stpipe-log.cfg file is:

```
[*]
handler = file:pipeline.log
level = INFO
```

If there's no `stpipe-log.cfg` file in the working directory, which specifies how to handle process log information, the default is to display log messages to stdout.

For example:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
    --logcfg=pipeline-log.cfg
```

Or in an interactive python environment:

```
result = Detector1Pipeline.call("jw00017001001_01101_00001_nrca1_uncal.fits",
                                logcfg="pipeline-log.cfg")
```

and the file `pipeline-log.cfg` contains:

```
[*]
handler = file:pipeline.log
level = INFO
```

In this example log information is written to a file called `pipeline.log`. The `level` argument in the log cfg file can be set to one of the standard logging level designations of `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. Only messages at or above the specified level will be displayed.

Note: Setting up `stpipe-log.cfg` can lead to confusion, especially if it is forgotten about. If one has not run a pipeline in awhile, and then sees no logging information, most likely it is because `stpipe-log.cfg` is present. Consider using a different name and specifying it explicitly on the command line.

CHAPTER

ELEVEN

JWST DATAMODELS

The `jwst` package also contains the interface for JWST Datamodels. `Datamodels` are the recommended way of reading and writing JWST data files (.fits) and reference files (.fits and .asdf). JWST data are encoded in FITS files, and reference files consist of a mix of FITS and ASDF - datamodels were designed to abstract away these intricacies and provide a simple interface to the data. They represent the data in FITS extensions and meta data in FITS headers in a Python object with a tree-like structure. The following section gives a brief overview of `Datamodels` as they pertain to the pipeline - see Data Models (https://stdatamodels.readthedocs.io/en/latest/jwst/datamodels/index.html#data-models) for more detailed documentation on Datamodels.

11.1 Datamodels and the JWST pipeline

When *running the pipeline in python*, the inputs and outputs of running a pipeline or a step are JWST `Datamodels`.

The input to a pipeline/step can be a `Datamodel`, created from an input file on disk. E.g:

```
# running a single pipeline step, input is datamodel object
from jwst.linearity import LinearityStep
import stdatamodels.jwst.datamodels as dm
input_model = dm.open('jw00001001001_01101_00001_mirimage_uncal.fits')
result = LinearityStep.call(input_model)
```

If a string path to a file on disk is passed in, a `DataModel` object will be created internally when the pipeline/step is run.

By default, when running in Python, the corrected data will be returned in-memory as a `DataModel` instead of being written as an output file. See *controlling output file behavior* ` for instrucions on how to write the returned `DataModel` to an output file.

CHAPTER

TWELVE

CRDS PUB SERVER FREEZE AND DEPRECATION

12.1 Why and When

As of November 10, 2022, all observers should use the standard CRDS OPS server for JWST calibration reference files:

https://jwst-crds.stsci.edu

The PUB server:

https://jwst-crds-pub.stsci.edu

was set up in anticipation of rapid reference file updates during commissioning and Cycle 1. However, due to the trouble-free commissioning process, the smooth transition to science operations, and the subsequent confusion that has resulted from having two servers, it has been decided that the PUB server is no longer needed and will be decommissioned. To make this transition as smooth as possible, this update will take place in stages.

On November 10, 2022, all observers should begin to transition to using only the CRDS OPS server, https://jwst-crds.stsci.edu. See the *software documentation* for instructions about how to configure CRDS.

On December 2nd, access to the PUB server will no longer be available externally. The frozen PUB database will be maintained internally for 3 months. On March 1, the PUB server will be fully decommissioned and the institute will retain an internal archive of the maps and calibration reference files. Observers who wish to use historical files from the PUB server in the future will need to file a JWST Pipeline help desk ticket to access the information.

Part of the decommissioning process will include establishing guidance for how best to maintain reproducibility for new papers and for already-published papers that used the PUB server. This information will be included in a new JDox page, currently in preparation. Visit the JDox site (https://jwst-docs.stsci.edu/) for new information concerning JWST.

12.2 Transition Procedure

If using the PUB server, there are two simple tasks that need to be done to ensure a successful transition from using the PUB server to the JWST OPS server.

First, the folder containing the local CRDS cache, pointed to by the environment variable CRDS_PATH, should be cleared of all old CRDS information.

If created appropriately, the folder that CRDS_CACHE points to should contain ONLY CRDS content. The suggested way of ensuring a new, empty cache, is to create a new folder. For example, to create a CRDS cache folder under a user's home folder, using Linux, the command is:

```
$ mkdir $HOME/crds_cache
```

Then set CRDS_PATH to point to this new, empty folder:

```
$ export CRDS_PATH=$HOME/crds_cache
```

The important point is that whatever folder is to be used to hold the CRDS cache should initially be empty; no other content should be present in the folder.

Older CRDS cache folders are no longer needed and can be removed as the user sees fit.

It does not matter what the folder is called, nor where it is located, as long as the user has access permissions to that folder. The location of the CRDS cache should contain sufficient space to hold the references. Current suggested minimum of free space is 100GB.

Second, ensure that the environment variable CRDS_SERVER_URL is pointing to the JWST OPS server, https://jwst-crds.stsci.edu:

```
$ export CRDS_SERVER_URL=https://jwst-crds.stsci.edu
```

Following these two steps ensures that further calibration processing will use references from the standard CRDS server.

CHAPTER

THIRTEEN

DATA PRODUCTS INFORMATION

13.1 Processing Levels and Product Stages

Here we describe the structure and content of the most frequently used forms of files for JWST science data products, the vast majority of which are in FITS format. Each type of FITS file is the result of serialization of a corresponding data model. All JWST pipeline input and output products, with the exception of a few reference files and catalogs, are serialized as FITS files. The ASDF (https://asdf-standard.readthedocs.io/en/stable/) representation of the data model is serialized as a FITS BINTABLE extension within the FITS file, with EXTNAME="ASDF". The ASDF extension is essentially a text character serialization in YAML (https://yaml.org) format of the data model. The ASDF representation is read from the extension when a FITS file is loaded into a data model to provide the initial instance of the data model. Values in the other FITS extensions then either override this initial model or are added to it.

Within the various STScI internal data processing and archiving systems that are used for routine processing of JWST data, there are some different uses of terminology to refer to different levels or stages of processing and products. For those who are interested or need to know, the table below gives high-level translations between those naming conventions.

Data Processing Levels	User Data Product Stages	MAST/CAOM Data Levels
N/A	N/A	-1 = Planned, but not yet executed
Level 0 = Science telemetry	Not available to users	Not available to users
Level 0.5 = POD files	Not available to users	Not available to users
Level 1a = Original FITS file	Stage 0 = Original FITS file	0 = raw (not available to users)
Level 1b = Uncal FITS file	Stage 0 = Fully-populated FITS file	1 = uncalibrated
Level 2a = Countrate exposure	Stage 1 = Countrate FITS file	2 = calibrated
Level 2b = Calibrated exposure *Level 2c = CR-flagged exposure	Stage 2 = Calibrated exposure	2 = calibrated
Level 3 = Combined data	Stage 3 = Combined data	3 = Science product

*Note that Level 2c files are intermediate files produced during pipeline Stage 3 processing, and are not final products (as opposed to all the other product types that are listed here). Therefore, Level 2c files are not a final product of any pipeline stage, but are produced within the pipeline Stage 3 processing. Level 2c files (identified by the 'crf' extension) are in the same format as Level 2b products, with the difference being that their data quality flags have been updated after running outlier detection in pipeline Stage 3 processing.

Throughout this document, we will use the "Stage" terminology to refer to data products. Stage 0, 1, and 2 products are always files containing the data from a single exposure and a single detector. A NIRCam exposure that uses all 10 detectors will therefore result in 10 separate FITS files for each of the Stage 0, 1, and 2 products. Because these stages

contain the data for a single exposure, they are refered to as "exposure-based" products and use an "exposure-based" file naming syntax. Stage 3 products, on the other hand, are constructed from the combined data of multiple exposures for a given source or target. They are referred to as "source-based" products and use a "source-based" file naming syntax. Observing modes that include multiple defined sources within a single exposure or observation, such as NIRSpec MOS and NIRCam/NIRISS WFSS, will result in multiple Stage 3 products, one for each defined or identifiable source.

13.2 File Naming Schemes

13.2.1 Exposure file names

The names of the exposure level data (stage 0 to 2) are constructed with information from the science header of the exposure, allowing users to map it to the observation in their corresponding APT files. The FITS file naming scheme for Stage 0, 1, and 2 "exposure-based" products is:

jw<ppppp><ooo><vvv>_<gg><s><aa>_<eeeee>(-<"seg"NNN>)_<detector>_<prodType>.fits

where

- ppppp: program ID number
- ooo: observation number
- vvv: visit number
- gg: visit group
- s: parallel sequence ID (1=prime, 2-5=parallel)
- aa: activity number (base 36)
- eeeee: exposure number
- segNNN: the text "seg" followed by a three-digit segment number (optional)
- detector: detector name (e.g. 'nrca1', 'nrcblong', 'mirimage')
- prodType: product type identifier (e.g. 'uncal', 'rate', 'cal')

An example Stage 1 product FITS file name is:

```
jw93065002001_02101_00001_nrca1_rate.fits
```

13.2.2 Stage 3 file names

In this stage, the calibration pipeline uses the association information to identify the relationship between exposures that are to be combined by design to form a single product. These data products result from the combination of multiple exposures like dithers or mosaics.

The format for the file name of a Stage 3 association product has all alphabetic characters in lower case, underscores are only used to delineate between major fields, and dashes can be used as separators for optional fields. Just as for Stage 2, the suffix distinguishes the different file products of Stage 3 of the calibration pipeline.

The FITS file naming scheme for Stage 3 "source-based" products is as follows, where items in parentheses are optional:

jw<ppppp>-<AC_ID>_[<"t"TargID | "s"SourceID>](-<"epoch"X>)_<instr>_<optElements>(-<subarray>)_<prodType>(-<ACT_ID>).fits

where

- ppppp: Program ID number

- AC_ID: Association candidate ID
- TargID: 3-digit Target ID (either TargID or SourceID must be present)
- SourceID: 5-digit Source ID
- epochX: The text "epoch" followed by a single digit epoch number (optional)
- instr: Science instrument name (e.g. 'nircam', 'miri')
- optElements: A single or hyphen-separated list of optical elements (e.g. filter, grating)
- subarray: Subarray name (optional)
- prodType: Product type identifier (e.g. 'i2d', 's3d', 'x1d')
- ACT_ID: 2-digit activity ID (optional)

An example Stage 3 product FITS file name is:

jw87600-a3001_t001_niriss_f480m-nrm_amiavg.fits

Optional Components

A number of components are optional, all either proposal-dependent or data-specific. The general cases where an optional component may appear are as follows:

TargID vs SourceID

For single-target modes, this is the target identifier as defined in the APT proposal.

For multi-object modes, such as NIRSpec MOS, this will be the slit ID for each object.

epochX

If a proposal has specified that observations be performed in multiple epochs, this will be the epoch id.

subarray

Present for all instruments/observing modes that allow subarray specification.

ACT_ID

Present when associations are dependent on being unique across visit activities. Currently, only Wavefront Sensing & Control (WFS&C) coarse and fine phasing are activity-dependent.

13.2.3 Segmented Products

When the raw data volume for an individual exposure is determined to be large enough to result in Stage 2 products greater than 2 GB in size, all Stage 0-2 products for the exposure are broken into multiple segments, so as to keep total file sizes to a reasonable level. This is often the case with Time Series Observations (TSO), where individual exposures can have thousands of integrations. The detector data are broken along integration boundaries (never within an integration) and stored in "segmented" products. The segments are identified by the "segNNN" field in exposure-based file names, where NNN is 1-indexed and always includes any leading zeros.

Segmented products contain extra keywords located in their primary headers that help to identify the segments and the contents of each segment. The following segment-related keywords are used:

- EXSEGNUM: The segment number of the current product
- EXSEGTOT: The total number of segments
- INTSTART: The starting integration number of the data in this segment

- INTEND: The ending integration number of the data in this segment

All of the Stage 1 and Stage 2 calibration pipelines will process each segment independently, creating the full set of intermediate and calibrated products for each segment. The calibrated data for all segments is then combined by one of the Stage 3 pipelines into a source-based Stage 3 product.

13.3 Data Product Types

The following tables contain lists of all data product types, as given by their file name suffix. There is one table per stage of processing. All tables indicate whether the file naming is exposure-based (Exp) or source-based (Src). When the product is not created by default, the flag *Optional* is indicated in the description. The different stages of the calibration pipeline are as defined in the Algorithms Documentation (https://jwst-docs.stsci.edu/jwst-data-reduction-pipeline/algorithm-documentation). The product name suffixes are active links to detailed descriptions in the following sections.

13.3.1 Stage 0 and Stage 1 Data Products

Pipeline	Input	Output(s)	Stage	Base	Units	Description
N/A		*uncal*	0	Exp	DN	Uncalibrated 4-D exposure data
calwebb_dark	*uncal*	*dark*	1	Exp	DN	4-D corrected dark exposure data
calwebb_detector1	*uncal*	*trapsfilled*	1	Exp	N/A	Charge trap state data
		rateints			DN/s	3-D countrate data (per integration)
		rate				2-D countrate data (per exposure)
		fitopt			various	*Optional* fit info from ramp_fit step
		dark			DN	*Optional* 3-D on-the-fly dark data
		ramp				*Optional* 4-D corrected ramp data

13.3.2 Stage 2 Data Products

Pipeline	Input	Output(s)	Base	Units	Description
calwebb_image2	*rate*	*bsub*	Exp	DN/s	2-D background-subtracted data, when background step applied
		cal		MJy/sr, MJy^2	2-D calibrated data
		i2d			2-D resampled imaging data
calwebb_image2 with TSO data	*rateints*	*calints*		MJy/sr, MJy[Page 49, 2]	3-D calibrated data; coronagraphy and TSO
calwebb_spec2	*rate*	*bsub*	Exp	DN/s	2-D background-subtracted data, when background step applied
		cal		MJy/sr, MJy[Page 49, 2]	2-D calibrated data
		s3d			3-D resampled spectroscopic data; NIRSpec IFU and MIRI MRS
		s2d			2-D resampled spectroscopic data
		x1d		various	1-D extracted spectral data

[2] NIRSpec and NIRISS SOSS point sources have MJy units; all others are MJy/sr

13.3.3 Stage 3 Data Products

Pipeline	Input	Outputs	Base	Units	Description
calwebb_image3	*cal*	*crf*	Exp	MJy/sr, MJy[Page 49, 2]	2-D CR-flagged calibrated data
		i2d	Src		2-D resampled imaging data
		cat		N/A	Source catalog
		segm		N/A	Segmentation map
calwebb_spec3	*cal*	*crf*	Exp	MJy/sr, MJy[Page 49, 2]	2-D CR-flagged calibrated data
		s2d	Src		2-D resampled spectroscopic data; Non-IFU
		s3d			3-D resampled spectroscopic data; NIRSpec IFU and MIRI MRS
		x1d		various	1-D extracted spectroscopic data
		c1d		various	1-D combined spectroscopic data
calwebb_ami3	*cal*	*ami*	Exp	N/A	Fringe parameters (per exposure)

13.4 Common Features

All JWST FITS data products have a few common features in their structure and organization:

1. The FITS primary Header Data Unit (HDU) only contains header information, in the form of keyword records, with an empty data array, which is indicated by the occurrence of NAXIS=0 in the primary header. Meta data that pertains to the entire product is stored in keywords in the primary header. Meta data related to specific extensions (see below) is stored in keywords in the headers of each extension.
2. All data related to the product are contained in one or more FITS IMAGE or BINTABLE extensions. The header of each extension may contain keywords that pertain uniquely to that extension.

13.5 Science products

The following sections describe the format and contents of each of the JWST FITS science products. Things to note in the descriptions include:

- Not all FITS extensions occur in every data product of a given type. Many are either optional or dependent on the instrument or observing mode. Such optional extensions are noted with an asterisk in the tables below.
- Because some extensions are optional, as well as the fact that the exact ordering of the extensions is not guaranteed, the FITS HDU index numbers of a given extension type can vary from one product to another. The only guarantee is that the `SCI` extension, containing the actual pixel values, will always be the first FITS extension (HDU=1). Other common extensions, like `DQ` and `ERR`, usually immediately follow the `SCI`, but the order is not guaranteed. Hence HDU index numbers are not listed for many extension types, because they can vary.

13.5.1 Uncalibrated raw data: `uncal`

Exposure raw data products are designated by a file name suffix of "uncal." These files usually contain only the raw detector pixel values from an exposure, with the addition of some table extensions containing various types of meta data associated with the exposure. Additional extensions can be included for certain instruments and readout types, as noted below. The FITS file structure is as follows.

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	uint16	ncols x nrows x ngroups x nints
2	GROUP	BINTABLE	N/A	variable
3	INT_TIMES	BINTABLE	N/A	nints (rows) x 7 cols
	ZEROFRAME*	IMAGE	uint16	ncols x nrows x nints
	REFOUT*	IMAGE	uint16	ncols/4 x nrows x ngroups x nints
	ASDF	BINTABLE	N/A	variable

- SCI: 4-D data array containing the raw pixel values. The first two dimensions are equal to the size of the detector readout, with the data from multiple groups (NGROUPS) within each integration stored along the 3rd axis, and the multiple integrations (NINTS) stored along the 4th axis.
- GROUP: A table of meta data for some (or all) of the data groups.
- INT_TIMES: A table of beginning, middle, and end time stamps for each integration in the exposure.
- ZEROFRAME: 3-D data array containing the pixel values of the zero-frame for each integration in the exposure, where each plane of the cube corresponds to a given integration. Only appears if the zero-frame data were requested to be downlinked separately.

- REFOUT: The MIRI detector reference output values. Only appears in MIRI exposures.
- ADSF: The data model meta data.

This FITS file structure is the result of serializing a `Level1bModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.Lev but can also be read into a `RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamod in which case zero-filled ERR, GROUPDQ, and PIXELDQ data arrays will be created and stored in the model, having array dimensions based on the shape of the SCI array (see `RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamodels.RampModel)).

13.5.2 Ramp data: `ramp`

As raw data progress through the *calwebb_detector1* pipeline they are stored internally in a `RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamodels.RampModel). This type of data model is serialized to a `ramp` type FITS file on disk. The original detector pixel values (in the SCI extension) are converted from integer to floating-point data type. The same is true for the ZEROFRAME and REFOUT data extensions, if they are present. An ERR array and two types of data quality arrays are also added to the product. The FITS file layout is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x ngroups x nints
2	PIXELDQ	IMAGE	uint32	ncols x nrows
3	GROUPDQ	IMAGE	uint8	ncols x nrows x ngroups x nints
4	ERR	IMAGE	float32	ncols x nrows x ngroups x nints
	ZEROFRAME*	IMAGE	float32	ncols x nrows x nints
	GROUP	BINTABLE	N/A	variable
	INT_TIMES	BINTABLE	N/A	nints (rows) x 7 cols
	REFOUT*	IMAGE	uint16	ncols/4 x nrows x ngroups x nints
	ASDF	BINTABLE	N/A	variable

- SCI: 4-D data array containing the pixel values. The first two dimensions are equal to the size of the detector readout, with the data from multiple groups (NGROUPS) within each integration stored along the 3rd axis, and the multiple integrations (NINTS) stored along the 4th axis.
- PIXELDQ: 2-D data array containing DQ flags that apply to all groups and all integrations for a given pixel (e.g. a hot pixel is hot in all groups and integrations).
- GROUPDQ: 4-D data array containing DQ flags that pertain to individual groups within individual integrations, such as the point at which a pixel becomes saturated within a given integration.
- ERR: 4-D data array containing uncertainty estimates on a per-group and per-integration basis.
- ZEROFRAME: 3-D data array containing the pixel values of the zero-frame for each integration in the exposure, where each plane of the cube corresponds to a given integration. Only appears if the zero-frame data were requested to be downlinked separately.
- GROUP: A table of meta data for some (or all) of the data groups.
- INT_TIMES: A table of beginning, middle, and end time stamps for each integration in the exposure.
- REFOUT: The MIRI detector reference output values. Only appears in MIRI exposures.
- ADSF: The data model meta data.

13.5.3 Countrate data: `rate` and `rateints`

Countrate products are produced by applying the *ramp_fitting* step to the integrations within an exposure, in order to compute count rates from the original accumulating signal ramps. For exposures that contain multiple integrations (NINTS > 1) this is done in two ways, which results in two separate products. First, countrates are computed for each integration within the exposure, the results of which are stored in a `rateints` product. These products contain 3-D data arrays, where each plane of the data cube contains the countrate image for a given integration.

The results for each integration are also averaged together to form a single 2-D countrate image for the entire exposure. These results are stored in a `rate` product.

The FITS file structure for a `rateints` product is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x nints
2	ERR	IMAGE	float32	ncols x nrows x nints
3	DQ	IMAGE	uint32	ncols x nrows x nints
4	INT_TIMES	BINTABLE	N/A	nints (rows) x 7 cols
5	VAR_POISSON	IMAGE	float32	ncols x nrows x nints
6	VAR_RNOISE	IMAGE	float32	ncols x nrows x nints
7	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of DN/s. The first two dimensions are equal to the size of the detector readout, with the data from multiple integrations stored along the 3rd axis.
- ERR: 3-D data array containing uncertainty estimates on a per-integration basis. These values are based on the combined VAR_POISSON and VAR_RNOISE data (see below), given as standard deviation.
- DQ: 3-D data array containing DQ flags. Each plane of the cube corresponds to a given integration.
- INT_TIMES: A table of beginning, middle, and end time stamps for each integration in the exposure.
- VAR_POISSON: 3-D data array containing the per-integration variance estimates for each pixel, based on Poisson noise only.
- VAR_RNOISE: 3-D data array containing the per-integration variance estimates for each pixel, based on read noise only.
- ADSF: The data model meta data.

These FITS files are compatible with the `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html
data model.

The FITS file structure for a `rate` product is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows
2	ERR	IMAGE	float32	ncols x nrows
3	DQ	IMAGE	uint32	ncols x nrows
4	VAR_POISSON	IMAGE	float32	ncols x nrows x nints
5	VAR_RNOISE	IMAGE	float32	ncols x nrows x nints
6	ASDF	BINTABLE	N/A	variable

- SCI: 2-D data array containing the pixel values, in units of DN/s.

- ERR: 2-D data array containing uncertainty estimates for each pixel. These values are based on the combined VAR_POISSON and VAR_RNOISE data (see below), given as standard deviation.
- DQ: 2-D data array containing DQ flags for each pixel.
- VAR_POISSON: 2-D data array containing the variance estimate for each pixel, based on Poisson noise only.
- VAR_RNOISE: 2-D data array containing the variance estimate for each pixel, based on read noise only.
- ADSF: The data model meta data.

These FITS files are compatible with the `ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.htr data model.

Note that the `INT_TIMES` table does not appear in `rate` products, because the data have been averaged over all integrations and hence the per-integration time stamps are no longer relevant.

13.5.4 Background-subtracted data: `bsub` and `bsubints`

The *calwebb_image2* and *calwebb_spec2* pipelines have the capability to perform background subtraction on countrate data. In its simplest form, this consists of subtracting background exposures or a CRDS background reference image from science images. This operation is performed by the *background* step in the stage 2 pipelines. If the pipeline parameter `save_bsub` is set to `True`, the result of the background subtraction step will be saved to a file. Because this is a direct image-from-image operation, the form of the result is identical to input. If the input is a `rate` product, the background-subtracted result will be a `bsub` product, which has the exact same structure as the *rate* product described above. Similarly, if the input is a `rateints` product, the background-subtracted result will be saved to a `bsubints` product, with the exact same structure as the *rateints* product described above.

13.5.5 Calibrated data: `cal` and `calints`

Single exposure calibrated products duplicate a lot of the format and content of countrate products. There are two different high-level forms of calibrated products: one containing results for all integrations in an exposure (`calints`) and one for results averaged over all integrations (`cal`). These products are the main result of Stage 2 pipelines like *calwebb_image2* and *calwebb_spec2*. There are many additional types of extensions that only appear for certain observing modes or instruments, especially for spectroscopic exposures.

The FITS file structure for a `calints` product is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x nints
2	ERR	IMAGE	float32	ncols x nrows x nints
3	DQ	IMAGE	uint32	ncols x nrows x nints
	INT_TIMES	BINTABLE	N/A	nints (rows) x 7 cols
	VAR_POISSON	IMAGE	float32	ncols x nrows x nints
	VAR_RNOISE	IMAGE	float32	ncols x nrows x nints
	VAR_FLAT	IMAGE	float32	ncols x nrows x nints
	AREA*	IMAGE		ncols x nrows
	WAVELENGTH*	IMAGE	float32	ncols x nrows
	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of surface brightness, for each integration.
- ERR: 3-D data array containing uncertainty estimates for each pixel, for each integration. These values are based on the combined VAR_POISSON and VAR_RNOISE data (see below), given as standard deviation.

- DQ: 3-D data array containing DQ flags for each pixel, for each integration.
- INT_TIMES: A table of beginning, middle, and end time stamps for each integration in the exposure.
- VAR_POISSON: 3-D data array containing the variance estimate for each pixel, based on Poisson noise only, for each integration.
- VAR_RNOISE: 3-D data array containing the variance estimate for each pixel, based on read noise only, for each integration.
- VAR_FLAT: 2-D data array containing the variance estimate for each pixel, based on uncertainty in the flat-field.
- AREA: 2-D data array containing pixel area values, added by the *photom* step, for imaging modes.
- WAVELENGTH: 2-D data array of wavelength values for each pixel, for some spectroscopic modes.
- ADSF: The data model meta data.

The FITS file structure for a `cal` product is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows
2	ERR	IMAGE	float32	ncols x nrows
3	DQ	IMAGE	uint32	ncols x nrows
4	VAR_POISSON	IMAGE	float32	ncols x nrows
5	VAR_RNOISE	IMAGE	float32	ncols x nrows
6	VAR_FLAT	IMAGE	float32	ncols x nrows
	AREA*	IMAGE	float32	ncols x nrows
	WAVELENGTH*	IMAGE	float32	ncols x nrows
	PATHLOSS_PS*	IMAGE	float32	ncols x nrows
	PATHLOSS_UN*	IMAGE	float32	ncols x nrows
	BARSHADOW*	IMAGE	float32	ncols x nrows
	ASDF	BINTABLE	N/A	variable

- SCI: 2-D data array containing the pixel values, in units of surface brightness.
- ERR: 2-D data array containing uncertainty estimates for each pixel. These values are based on the combined VAR_POISSON and VAR_RNOISE data (see below), given as standard deviation.
- DQ: 2-D data array containing DQ flags for each pixel.
- VAR_POISSON: 2-D data array containing the variance estimate for each pixel, based on Poisson noise only.
- VAR_RNOISE: 2-D data array containing the variance estimate for each pixel, based on read noise only.
- VAR_FLAT: 2-D data array containing the variance estimate for each pixel, based on uncertainty in the flat-field.
- AREA: 2-D data array containing pixel area values, added by the *photom* step, for imaging modes.
- WAVELENGTH: 2-D data array of wavelength values for each pixel, for some spectroscopic modes.
- PATHLOSS_PS: 2-D data array of point-source pathloss correction factors, added by the *pathloss* step, for some spectroscopic modes.
- PATHLOSS_UN: 1-D data array of uniform-source pathloss correction factors, added by the *pathloss* step, for some spectroscopic modes.
- BARSHADOW: 2-D data array of NIRSpec MSA bar shadow correction factors, added by the *barshadow* step, for NIRSpec MOS exposures only.
- ADSF: The data model meta data.

For spectroscopic modes that contain data for multiple sources, such as NIRSpec MOS, NIRCam WFSS, and NIRISS WFSS, there will be multiple tuples of the SCI, ERR, DQ, VAR_POISSON, VAR_RNOISE, etc. extensions, where each tuple contains the data for a given source or slit, as created by the *extract_2d* step. FITS "EXTVER" keywords are used in each extension header to segregate the multiple instances of each extension type by source.

13.5.6 Cosmic-Ray flagged data: `crf` and `crfints`

Several of the stage 3 pipelines, such as *calwebb_image3* and *calwebb_spec3*, include the *outlier detection* step, which finds and flags outlier pixel values within calibrated images. The results of this process have the identical format and content as the input `cal` and `calints` products. The only difference is that the DQ arrays have been updated to contain CR flags. If the inputs are in the form of `cal` products, the CR-flagged data will be saved to a `crf` product, which has the exact same structure and content as the *cal* product described above. Similarly, if the inputs are `calints` products, the CR-flagged results will be saved to a `crfints` product, which has the same structure and content as the *calints* product described above.

13.5.7 Resampled 2-D data: `i2d` and `s2d`

Images and spectra that have been resampled by the *resample* step use a different set of data arrays than other science products. Resampled 2-D images are stored in `i2d` products and resampled 2-D spectra are stored in `s2d` products. The FITS file structure for `i2d` and `s2d` products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows
2	ERR	IMAGE	float32	ncols x nrows
3	CON	IMAGE	int32	ncols x nrows x nplanes
4	WHT	IMAGE	float32	ncols x nrows
5	VAR_POISSON	IMAGE	float32	ncols x nrows
6	VAR_RNOISE	IMAGE	float32	ncols x nrows
7	VAR_FLAT	IMAGE	float32	ncols x nrows
	HDRTAB*	BINTABLE	N/A	variable
	ASDF	BINTABLE	N/A	variable

- SCI: 2-D data array containing the pixel values, in units of surface brightness
- ERR: 2-D data array containing resampled uncertainty estimates, given as standard deviation
- CON: 3-D context image, which encodes information about which input images contribute to a specific output pixel
- WHT: 2-D weight image giving the relative weight of the output pixels
- VAR_POISSON: 2-D resampled Poisson variance estimates for each pixel
- VAR_RNOISE: 2-D resampled read noise variance estimates for each pixel
- VAR_FLAT: 2-D resampled flat-field variance estimates for each pixel
- HDRTAB: A table containing meta data (FITS keyword values) for all of the input images that were combined to produce the output image. Only appears when multiple inputs are used.
- ADSF: The data model meta data.

For spectroscopic exposure-based products that contain spectra for more than one source or slit (e.g. NIRSpec MOS) there will be multiple tuples of the SCI, ERR, CON, WHT, and variance extensions, one set for each source or slit.

FITS "EXTVER" keywords are used in each extension header to segregate the multiple instances of each extension type by source.

For the context array, CON, though the schema represents it as an `int32`, users should interpret and recast the array as `uint32` post-processing. This inconsistency will be dealt with in a later release.

13.5.8 Resampled 3-D (IFU) data: `s3d`

3-D IFU cubes created by the *cube_build* step are stored in FITS files with the following structure:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x nwaves
2	ERR	IMAGE	float32	ncols x nrows x nwaves
3	DQ	IMAGE	uint32	ncols x nrows x nwaves
4	WMAP	IMAGE	float32	ncols x nrows x nwaves
	WCS-TABLE	BINTABLE	N/A	2 cols x 1 row
	HDRTAB*	BINTABLE	N/A	variable
	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the spaxel values, in units of surface brightness.
- ERR: 3-D data array containing uncertainty estimates for each spaxel.
- DQ: 3-D data array containing DQ flags for each spaxel.
- WMAP: 3-D weight image giving the relative weights of the output spaxels.
- WCS-TABLE: A table listing the wavelength to be associated with each plane of the third axis in the SCI, DQ, ERR, and WMAP arrays, in a format that conforms to the FITS spectroscopic WCS standards. Column 1 of the table ("nelem") gives the number of wavelength elements listed in the table and column 2 ("wavelength") is a 1-D array giving the wavelength values.
- HDRTAB: A table containing meta data (FITS keyword values) for all of the input images that were combined to produce the output image. Only appears when multiple inputs are used.
- ADSF: The data model meta data.

`s3d` products contain several unique meta data elements intended to aid in the use of these products in data analysis tools. This includes the following keywords located in the header of the FITS primary HDU:

- FLUXEXT: A string value containing the EXTNAME of the extension containing the IFU flux data. Normally set to "SCI" for JWST IFU cube products.
- ERREXT: A string value containing the EXTNAME of the extension containing error estimates for the IFU cube. Normally set to "ERR" for JWST IFU cube products.
- ERRTYPE: A string value giving the type of error estimates contained in ERREXT, with possible values of "ERR" (error = standard deviation), "IERR" (inverse error), "VAR" (variance), and "IVAR" (inverse variance). Normally set to "ERR" for JWST IFU cube products.
- MASKEXT: A string value containing the EXTNAME of the extension containing the Data Quality mask for the IFU cube. Normally set to "DQ" for JWST IFU cube products.

In addition, the following WCS-related keywords are included in the header of the "SCI" extension to support the use of the wavelength table contained in the "WCS-TABLE" extension. These keywords allow data analysis tools that are compliant with the FITS spectroscopic WCS standards to automatically recognize and load the wavelength information in the "WCS-TABLE" and assign wavelengths to the IFU cube data.

- PS3_0 = 'WCS-TABLE': The name of the extension containing coordinate data for axis 3.
- PS3_1 = 'wavelength': The name of the table column containing the coordinate data.

The coordinate data (wavelength values in this case) contained in the "WCS-TABLE" override any coordinate information normally computed from FITS WCS keywords like CRPIX3, CRVAL3, and CDELT3 for coordinate axis 3.

13.5.9 Extracted 1-D spectroscopic data: `x1d` and `x1dints`

Extracted spectral data produced by the *extract_1d* step are stored in binary table extensions of FITS files. The overall layout of the FITS file is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	EXTRACT1D	BINTABLE	N/A	variable
2	ASDF	BINTABLE	N/A	variable

- EXTRACT1D: A 2-D table containing the extracted spectral data.
- ADSF: The data model meta data.

Multiple "EXTRACT1D" extensions can be present if there is data for more than one source or if the file is an `x1dints` product. For `x1dints` products, there is one "EXTRACT1D" extension for each integration in the exposure.

The structure of the "EXTRACT1D" table extension is as follows:

Column Name	Data Type	Contents | Units	
WAVELENGTH	float64	Wavelength values | μ m	
FLUX	float64	Flux values	Jy
FLUX_ERROR	float64	Error values	Jy
FLUX_VAR_POISSON	float64	Error values	Jy^2
FLUX_VAR_RNOISE	float64	Error values	Jy^2
FLUX_VAR_FLAT	float64	Error values	Jy^2
SURF_BRIGHT	float64	Surface Brightness	MJy/sr
SB_ERROR	float64	Surf. Brt. errors	MJy/sr
SB_VAR_POISSON	float64	Surf. Brt. errors	(MJy/sr)^2
SB_VAR_RNOISE	float64	Surf. Brt. errors	(MJy/sr)^2
SB_VAR_FLAT	float64	Surf. Brt. errors	(MJy/sr)^2
DQ	uint32	DQ flags	N/A
BACKGROUND	float64	Background signal	MJy/sr
BKGD_ERROR	float64	Background error	MJy/sr
BKGD_VAR_POISSON	float64	Background error	(MJy/sr)^2
BKGD_VAR_RNOISE	float64	Background error	(MJy/sr)^2
BKGD_VAR_FLAT	float64	Background error	(MJy/sr)^2
NPIXELS	float64	Number of pixels	N/A

The table is constructed using a simple 2-D layout, using one row per extracted spectral element in the dispersion direction of the data (i.e. one row per wavelength bin). Note that for point sources observed with NIRSpec or NIRISS SOSS mode, it is not possible to express the extracted spectrum as surface brightness and hence the SURF_BRIGHT and SB_ERROR columns will be set to zero. NPIXELS gives the (fractional) number of pixels included in the source extraction region at each wavelength bin.

13.5.10 Combined 1-D spectroscopic data: `c1d`

Combined spectral data produced by the *combine_1d* step are stored in binary table extensions of FITS files. The overall layout of the FITS file is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	COMBINE1D	BINTABLE	N/A	variable
2	ASDF	BINTABLE	N/A	variable

- COMBINE1D: A 2-D table containing the combined spectral data.
- ADSF: The data model meta data.

The structure of the "COMBINE1D" table extension is as follows:

Column Name	Data Type	Contents	Units
WAVELENGTH	float64	Wavelength values	μ m
FLUX	float64	Flux values	Jy
ERROR	float64	Error values	Jy
SURF_BRIGHT	float64	Surface Brightness	MJy/sr
SB_ERROR	float64	Surf. Brt. errors	MJy/sr
DQ	uint32	DQ flags	N/A
WEIGHT	float64	Sum of weights	N/A
N_INPUT	float64	Number of inputs	N/A

The table is constructed using a simple 2-D layout, using one row per extracted spectral element in the dispersion direction of the data (i.e. one row per wavelength bin).

13.5.11 Source catalog: `cat`

The *source_catalog* step contained in the *calwebb_image3* pipeline detects and quantifies sources within imaging products. The derived data for the sources is stored in a `cat` product, which is in the form of an ASCII table in ECSV (http://docs.astropy.org/en/stable/_modules/astropy/io/ascii/ecsv.html) (Enhanced Character Separated Values) format. It is a flat text file, containing meta data header entries and the source data in a 2-D table layout, with one row per source.

13.5.12 Segmentation map: `segm`

The *source_catalog* step contained in the *calwebb_image3* pipeline uses an image segmentation procedure to detect sources, which is a process of assigning a label to every image pixel that contains signal from a source, such that pixels belonging to the same source have the same label. The result of this procedure is saved in a `segm` product. The product is in FITS format, with a single image extension containing a 2-D image. The image has the same dimensions as the science image from which the sources were detected, and each pixel belonging to a source has an integer value corresponding to the label listed in the source catalog (`cat` product). Pixels not belonging to a source have a value of zero.

13.5.13 Photometry catalog: `phot`

The *tso_photometry* step in the *calwebb_tso3* pipeline produces light curve from TSO imaging observations by computing aperture photometry as a function of integration time stamp within one or more exposures. The resulting photometric data are stored in a `phot` product, which is in the form of an ASCII table in ECSV (http://docs.astropy.org/en/stable/_modules/astropy/io/ascii/ecsv.html) (Enhanced Character Separated Values) format. It is a flat text file, containing meta data header entries and the photometric data in a 2-D table layout, with one row per exposure integration.

13.5.14 White-light photometric timeseries: `whtlt`

The *white_light* step in the *calwebb_tso3* pipeline produces a light curve from TSO spectroscopic observations by computing the wavelength-integrated spectral flux as a function of integration time stamp within one or more exposures. The resulting photometric timeseries data are stored in a `whtlt` product, which is in the form of an ASCII table in ECSV (http://docs.astropy.org/en/stable/_modules/astropy/io/ascii/ecsv.html) (Enhanced Character Separated Values) format. It is a flat text file, containing meta data header entries and the white-light flux data in a 2-D table layout, with one row per exposure integration.

13.5.15 Stacked PSF data: `psfstack`

The *stack_refs* step in the *calwebb_coron3* pipeline takes a collection of PSF reference image and assembles them into a 3-D stack of PSF images, which results in a `psfstack` product. The `psfstack` product uses the `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeModel) data model, which when serialized to a FITS file has the structure shown below.

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x npsfs
2	DQ	IMAGE	uint32	ncols x nrows x npsfs
3	ERR	IMAGE	float32	ncols x nrows x npsfs
4	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing a stack of 2-D PSF images.
- DQ: 3-D data array containing DQ flags for each PSF image.
- ERR: 3-D data array containing a stack of 2-D uncertainty estimates for each PSF image.
- ADSF: The data model meta data.

13.5.16 Aligned PSF data: `psfalign`

The *align_refs* step in the *calwebb_coron3* pipeline creates a 3-D stack of PSF images that are aligned to corresponding science target images. The resulting `psfalign` product uses the `QuadModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.QuadModel.html#jwst.datamodels.QuadModel) data model, which when serialized to a FITS file has the structure and content shown below.

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x npsfs x nints
2	DQ	IMAGE	uint32	ncols x nrows x npsfs x nints
3	ERR	IMAGE	float32	ncols x nrows x npsfs x nints
4	ASDF	BINTABLE	N/A	variable

- SCI: 4-D data array containing a stack of 2-D PSF images aligned to each integration within a corresponding science target exposure. each integration.
- DQ: 4-D data array containing DQ flags for each PSF image.
- ERR: 4-D data array containing a stack of 2-D uncertainty estimates for each PSF image, per science target integration.
- ADSF: The data model meta data.

13.5.17 PSF-subtracted data: `psfsub`

The *klip* step in the *calwebb_coron3* pipeline subtracts an optimized combination of PSF images from each integration in a science target exposure. The resulting PSF-subtracted science exposure data uses the `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeModel) data model, which when serialized to a FITS file has the structure shown below.

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x nints
2	ERR	IMAGE	float32	ncols x nrows x nints
3	DQ	IMAGE	uint32	ncols x nrows x nints
4	INT_TIMES	BINTABLE	N/A	nints (rows) x 7 cols
5	VAR_POISSON	IMAGE	float32	ncols x nrows x nints
6	VAR_RNOISE	IMAGE	float32	ncols x nrows x nints
7	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing a stack of 2-D PSF-subtracted science target images, one per integration.
- ERR: 3-D data array containing a stack of 2-D uncertainty estimates for each science target integration.
- DQ: 3-D data array containing DQ flags for each science target integration.
- INT_TIMES: A table of beginning, middle, and end time stamps for each integration in the exposure.
- VAR_POISSON: 3-D data array containing the per-integration variance estimates for each pixel, based on Poisson noise only.
- VAR_RNOISE: 3-D data array containing the per-integration variance estimates for each pixel, based on read noise only.
- ADSF: The data model meta data.

13.5.18 AMI data: `ami`, `amiavg`, and `aminorm`

AMI derived data created by the *ami_analyze*, *ami_average*, and *ami_normalize* steps, as part of the *calwebb_ami3* pipeline, are stored in FITS files that contain a mixture of images and binary table extensions. The output format of all three pipeline steps is the same, encapsulated within a `AmiLgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLgModel) data model. The overall layout of the corresponding FITS files is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	FIT	IMAGE	float32	ncols x nrows
2	RESID	IMAGE	float32	ncols x nrows
3	CLOSURE_AMP	BINTABLE	float64	1 col x 35 rows
4	CLOSURE_PHA	BINTABLE	float64	1 col x 35 rows
5	FRINGE_AMP	BINTABLE	float64	1 col x 21 rows
6	FRINGE_PHA	BINTABLE	float64	1 col x 21 rows
7	PUPIL_PHA	BINTABLE	float64	1 col x 7 rows
8	SOLNS	BINTABLE	float64	1 col x 44 rows
9	ASDF	BINTABLE	N/A	variable

- FIT: A 2-D image of the fitted model.
- RESID: A 2-D image of the fit residuals.
- CLOSURE_AMP: A table of closure amplitudes.
- CLOSURE_PHA: A table of closure phases.
- FRINGE_AMP: A table of fringe amplitudes.
- FRINGE_PHA: A table of fringe phases.
- PUPIL_PHA: A table of pupil phases.
- SOLNS: A table of fringe coefficients.
- ADSF: The data model meta data.

13.6 Non-science products

13.6.1 Dark exposure: `dark`

Dark exposures processed by the *calwebb_dark* pipeline result in a product that has the same structure and content as the *ramp* product described above. The details are as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x ngroups x nints
2	PIXELDQ	IMAGE	uint32	ncols x nrows
3	GROUPDQ	IMAGE	uint8	ncols x nrows x ngroups x nints
4	ERR	IMAGE	float32	ncols x nrows x ngroups x nints
5	GROUP	BINTABLE	N/A	variable
6	INT_TIMES	BINTABLE	N/A	nints (rows) x 7 cols
	ZEROFRAME*	IMAGE	float32	ncols x nrows x nints
	REFOUT*	IMAGE	uint16	ncols/4 x nrows x ngroups x nints
	ASDF	BINTABLE	N/A	variable

- SCI: 4-D data array containing the pixel values. The first two dimensions are equal to the size of the detector readout, with the data from multiple groups (NGROUPS) within each integration stored along the 3rd axis, and the multiple integrations (NINTS) stored along the 4th axis.
- PIXELDQ: 2-D data array containing DQ flags that apply to all groups and all integrations for a given pixel (e.g. a hot pixel is hot in all groups and integrations).
- GROUPDQ: 4-D data array containing DQ flags that pertain to individual groups within individual integrations, such as the point at which a pixel becomes saturated within a given integration.
- ERR: 4-D data array containing uncertainty estimates on a per-group and per-integration basis.
- GROUP: A table of meta data for some (or all) of the data groups.
- INT_TIMES: A table of beginning, middle, and end time stamps for each integration in the exposure.
- ZEROFRAME: 3-D data array containing the pixel values of the zero-frame for each integration in the exposure, where each plane of the cube corresponds to a given integration. Only appears if the zero-frame data were requested to be downlinked separately.
- REFOUT: The MIRI detector reference output values. Only appears in MIRI exposures.
- ADSF: The data model meta data.

13.6.2 Charge trap state data: `trapsfilled`

The *persistence* step in the *calwebb_detector1* pipeline produces an image containing information on the number of filled charge traps in each pixel at the end of an exposure. Internally these data exist as a `TrapsFilledModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.TrapsFilledModel.html#jwst.datamodels.TrapsFilledModel) data model, which is saved to a `trapsfilled` FITS product. The FITS file has the following format:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows x 3
2	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array giving the number of charge traps per pixel, with each plane corresponding to a different trap family.
- ADSF: The data model meta data.

13.6.3 WFS&C combined image: `wfscmb`

The *wfs_combine* step in the *calwebb_wfs-image3* pipeline combines dithered pairs of Wavefront Sensing and Control (WFS&C) images, with the result being stored in a `wfscmb` product. Unlike the drizzle methods used to combine and resample science images, resulting in an *i2d* product, the WFS&C combination is a simple shift and add technique that results in a standard imaging FITS file structure, as shown below.

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	ncols x nrows
2	ERR	IMAGE	float32	ncols x nrows
3	DQ	IMAGE	uint32	ncols x nrows
4	ASDF	BINTABLE	N/A	variable

- SCI: 2-D data array containing the pixel values, in units of surface brightness.
- ERR: 2-D data array containing uncertainty estimates for each pixel.
- DQ: 2-D data array containing DQ flags for each pixel.
- ADSF: The data model meta data.

13.7 Guide star data products

The FGS guiding capabilities are provided by 4 guide star functions: Identification, Acquisition, Track, and Fine Guide. Data downlinked by these functions is processed by DMS to provide uncalibrated and calibrated guide star data products. The uncalibrated products consist of raw pixel value data arrays, as well as different kinds of tabular information related to the guide stars and centroid locations of the guide star as computed by the on-board FGS Flight Software (FSW). Calibrated guide star products are created by the *calwebb_guider* pipeline. Briefly, the processing performed applies bad pixel masks and flat-fields the science data, as well as computing countrate images from the multiple groups within each integration contained in a given product. The countrate images, which are computed by the *guider_cds* step, are computed for most modes by simply differencing groups 1 and 2 in each integration and dividing by the group time.

13.7.1 File naming

Guide star product file names contain identifiers related to the function in use and the time at which the data were obtained. The table below lists the file name syntax used for each of the guiding functions and the related value of the EXP_TYPE keyword.

Function	EXP_TYPE	File name
Identification	FGS_ID-IMAGE	jw<pppppooovvv>_gs-id_<m>_image-uncal.fits
	FGS_ID-STACK	jw<pppppooovvv>_gs-id_<m>_stacked-uncal.fits
Acquisition	FGS_ACQ1	jw<pppppooovvv>_gs-acq1_<yyyydddhhmmss>_uncal.fits
	FGS_ACQ2	jw<pppppooovvv>_gs-acq2_<yyyydddhhmmss>_uncal.fits
Track	FGS_TRACK	jw<pppppooovvv>_gs-track_<yyyydddhhmmss>_uncal.fits
Fine Guide	FGS_FINEGUIDE	jw<pppppooovvv>_gs-fg_<yyyydddhhmmss>_uncal.fits

where the file name fields are:

jw
: mission identifier

ppppp
: program id

ooo
: observation number

vvv
: visit number

m
: ID attempt counter (1-8)

yyyydddhhmmss
: time stamp at the end of the data in the file

Uncalibrated products use the "uncal" file name suffix as shown above, while calibrated products use a "cal" suffix. The relevance of the "image" and "stacked" designations for the Identification mode products is described below.

13.7.2 ID mode

The "Identification" guiding function images the field of view by reading the detector in a series subarray "strips" that, collectively, cover most of the field. A total of 36 subarray strips are read out, each of which is 64 x 2048 pixels in size. Each strip has 8 pixels of overlap with its adjoining strips, resulting in a total of 2024 unique detector rows that've been read out. ID mode uses 2 groups per integration and 2 integrations, resulting in a total of 4 reads. Each subarray strip has its 4 reads performed before moving on to the next subarray.

DMS creates 2 different forms of products for ID mode data: one in which an image is constructed by simply stacking or butting the data from adjacent subarray strips against one another and the other in which the overlap regions of the strips are taken into account by averaging the pixel values. The first form is referred to as a "stacked" product and the second as an "image" product.

The FITS file structure for uncalibrated ID "image" products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	uint16	2024 x 2048 x 2 x 2
2	Flight Reference Stars	BINTABLE	N/A	4 cols x nstars
3	Planned Reference Stars	BINTABLE	N/A	10 cols x nstars

- SCI: 4-D data array containing the raw pixel values. The subarray overlaps have been accounted for, resulting in image dimensions of 2024 x 2048 pixels, with the 2 groups and 2 integrations stacked along the 3rd and 4th array axes.
- Flight Reference Stars: A table containing information on the actual reference stars used by the FSW. Detailed contents are listed *below*.
- Planned Reference Stars: A table containing information on the planned reference stars. Detailed contents are listed *below*.

The FITS file structure for uncalibrated ID "stacked" products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	uint16	2304 x 2048 x 2 x 2
2	Flight Reference Stars	BINTABLE	N/A	4 cols x nstars
3	Planned Reference Stars	BINTABLE	N/A	10 cols x nstars

- SCI: 4-D data array containing the raw pixel values. The subarray data are butted against one another, resulting in image dimensions of 2304 x 2048 pixels, with the 2 groups and 2 integrations stacked along the 3rd and 4th array axes.
- Flight Reference Stars: A table containing information on the actual reference stars used by the FSW. Detailed contents are listed *below*.
- Planned Reference Stars: A table containing information on the planned reference stars. Detailed contents are listed *below*.

The FITS file structure for calibrated ID "image" products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	2024 x 2048 x 1
2	ERR	IMAGE	float32	2024 x 2048 x 1
3	DQ	IMAGE	uint32	2024 x 2048
4	Flight Reference Stars	BINTABLE	N/A	4 cols x nstars
5	Planned Reference Stars	BINTABLE	N/A	10 cols x nstars
6	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of DN/s. The data for the 2 integrations has been combined into a single image, as is done by the on-board FSW, resulting in a data array with NAXIS3 = 1.
- ERR: 3-D data array containing uncertainty estimates for each pixel.
- DQ: 2-D data array containing DQ flags for each pixel.
- Flight Reference Stars: A table containing information on the actual reference stars used by the FSW. Detailed contents are listed *below*.
- Planned Reference Stars: A table containing information on the planned reference stars. Detailed contents are listed *below*.
- ADSF: The data model meta data.

The FITS file structure for calibrated ID "stacked" products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	2304 x 2048 x 1
2	ERR	IMAGE	float32	2304 x 2048 x 1
3	DQ	IMAGE	uint32	2304 x 2048
4	Flight Reference Stars	BINTABLE	N/A	4 cols x nstars
5	Planned Reference Stars	BINTABLE	N/A	10 cols x nstars
6	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of DN/s. The data for the 2 integrations has been combined into a single image, as is done by the on-board FSW, resulting in a data array with NAXIS3=1.

- ERR: 3-D data array containing uncertainty estimates for each pixel.
- DQ: 2-D data array containing DQ flags for each pixel.
- Flight Reference Stars: A table containing information on the actual reference stars used by the FSW. Detailed contents are listed *below*.
- Planned Reference Stars: A table containing information on the planned reference stars. Detailed contents are listed *below*.
- ADSF: The data model meta data.

Flight reference stars table

The structure and content of the Flight Reference Stars table is as follows.

Column Name	Data Type	Description
reference_star_id	char*2	Reference star index
id_x	float64	x position in FGS Ideal frame
id_y	float64	y position in FGS Ideal frame
count_rate	float64	count rate

Planned reference stars table

The structure and content of the Planned Reference Stars table is as follows.

Column Name	Data Type	Description
guide_star_order	int32	Guide star index within list
reference_star_id	char*12	GSC II identifier
ra	float64	ICRS RA of the star
dec	float64	ICRS Dec of the star
id_x	float64	x position in FGS Ideal frame
id_y	float64	y position in FGS Ideal frame
fgs_mag	float64	magnitude
fgs_mag_uncert	float64	magnitude uncertainty
count_rate	float64	count rate
count_rate_uncert	float64	count rate uncertainty

13.7.3 ACQ1 mode

The "Acquisition" guiding function ACQ1 performs 128 x 128 pixel subarray readouts of the detector, using 2 groups per integration and a total of 6 integrations. The FITS file structure for ACQ1 uncalibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	uint16	128 x 128 x 2 x 6

- SCI: 4-D data array containing the raw pixel values.

The FITS file structure for ACQ1 calibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	128 x 128 x 6
2	ERR	IMAGE	float32	128 x 128 x 6
3	DQ	IMAGE	uint32	128 x 128
4	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of DN/s. Count rate images have been computed for each of the 6 integrations by differencing the 2 groups of each integration.
- ERR: 3-D data array containing uncertainty estimates for each pixel.
- DQ: 2-D data array containing DQ flags for each pixel.
- ADSF: The data model meta data.

13.7.4 ACQ2 mode

The "Acquisition" guiding function ACQ2 performs 32 x 32 pixel subarray readouts of the detector, using 2 groups per integration and a total of 5 integrations. The FITS file structure for ACQ2 uncalibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	uint16	32 x 32 x 2 x 5

- SCI: 4-D data array containing the raw pixel values.

The FITS file structure for ACQ2 calibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	32 x 32 x 5
2	ERR	IMAGE	float32	32 x 32 x 5
3	DQ	IMAGE	uint32	32 x 32
4	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of DN/s. Count rate images have been computed for each of the 5 integrations by differencing the 2 groups of each integration.
- ERR: 3-D data array containing uncertainty estimates for each pixel.
- DQ: 2-D data array containing DQ flags for each pixel.
- ADSF: The data model meta data.

13.7.5 Track mode

The "Track" guiding function performs 32 x 32 pixel subarray readouts, the location of which can move on the detector as the FGS FSW tracks the position of the guide star. The subarray readouts are performed with a cadence of 16 Hz. Each integration consists of 2 groups, and the total number of integrations (NINTS) can be very large (in the thousands). The FITS file structure for TRACK uncalibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	uint16	32 x 32 x 2 x nints
2	Pointing	BINTABLE	N/A	12 cols x nrows
3	FGS Centroid Packet	BINTABLE	N/A	17 cols x nrows
4	Track subarray table	BINTABLE	N/A	5 cols x nrows

- SCI: 4-D data array containing the raw pixel values.
- Pointing: A table containing guide star position and jitter information. See *below* for details of the contents.
- FGS Centroid Packet: A table containing guide star centroiding information. See *below* for details of the contents.
- Track subarray table: A table containing subarray information over the duration of the product. See *below* for details of the contents.

The FITS file structure for TRACK calibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	32 x 32 x nints
2	ERR	IMAGE	float32	32 x 32 x nints
3	DQ	IMAGE	uint32	32 x 32
4	POINTING	BINTABLE	N/A	12 cols x nrows
5	FGS CENTROID PACKET	BINTABLE	N/A	17 cols x nrows
6	TRACK SUBARRAY TABLE	BINTABLE	N/A	5 cols x nrows
7	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of DN/s. Count rate images for each integration have been computed by differencing the 2 groups in each integration.
- ERR: 3-D data array containing uncertainty estimates for each pixel.
- DQ: 2-D data array containing DQ flags for each pixel.
- Pointing: A table containing guide star position and jitter information. See *below* for details of the contents.
- FGS Centroid Packet: A table containing guide star centroiding information. See *below* for details of the contents.
- Track subarray table: A table containing subarray information over the duration of the product. See *below* for details of the contents.
- ADSF: The data model meta data.

Pointing table

The structure and content of the Pointing table is as follows.

Column Name	Data Type	Units	Description
time	float64	milli-sec	Time since start of data file
jitter	float64	milli-arcsec	$sqrt(delta\_ddc\_ra^2 + delta\_ddc\_dec^2)$
delta_ddc_ra	float64	milli-arcsec	Initial DDC RA - Current
delta_ddc_dec	float64	milli-arcsec	Initial DDC Dec - Current
delta_aperture_pa	float64	milli-arcsec	Initial PA - Current
delta_v1_ra	float64	milli-arcsec	Initial V frame RA - Current
delta_v1_dec	float64	milli-arcsec	Initial V frame Dec - Current
delta_v3_pa	float64	milli-arcsec	Initial V frame PA - Current
delta_j1_ra	float64	milli-arcsec	Initial J frame RA - Current
delta_j1_dec	float64	milli-arcsec	Initial J frame Dec - Current
delta_j3_pa	float64	milli-arcsec	Initial J frame PA - Current
HGA_motion	int32	N/A	HGA state: 0 = moving, 1 = finished, 2 = offline

FGS Centroid Packet table

The structure and content of the Centroid Packet table is as follows.

Column Name	Data Type	Description
observatory_time	char*23	UTC time when packet was generated
centroid_time	char*23	Fine guidance centroid time
guide_star_position_x	float64	FGS Ideal Frame (arcsec)
guide_star_position_y	float64	FGS Ideal Frame (arcsec)
guide_star_instrument_counts_per_sec	float64	Instrument counts/sec
signal_to_noise_current_frame	float64	For current image frame
delta_signal	float64	Between current and previous frame
delta_noise	float64	Between current and previous frame
psf_width_x	int32	Bias from ideal guide star position (pixels)
psf_width_y	int32	Bias from ideal guide star position (pixels)
data_quality	int32	Centroid data quality
bad_pixel_flag	char*4	Bad pixel status for current subwindow (0/1)
bad_centroid_dq_flag	char*50	Bad centroid for current subwindow (0/1)
cosmic_ray_hit_flag	char*5	NO/YES
sw_subwindow_loc_change_flag	char*5	NO/YES
guide_star_at_detector_subwindow_boundary_flag	char*5	NO/YES
subwindow_out_of_FOV_flag	char*5	NO/YES

Track Subarray table

The Track Subarray table contains location and size information for the detector subarray window that is used during the track function to follow the guide star. The structure and content of the Track Subarray table is as follows.

Column Name	Data Type	Description
observatory_time	char*23	UTC time when packet was generated
x_corner	float64	Subarray x corner (pixels)
y_corner	float64	Subarray y corner (pixels)
x_size	int16	Subarray x size (pixels)
y_size	int16	Subarray y size (pixels)

13.7.6 FineGuide mode

The "FineGuide" guiding function performs 8 x 8 pixel subarray readouts, at a fixed location on the detector, and with a cadence of 16 Hz, from which the FGS FSW computes centroids for the guide star. To reduce readout noise contribution to the centroid calculation, "Fowler" sampling of the readouts is employed. Each integration consists of 4 readouts at the beginning, a signal accumulation period, and 4 readouts at the end. The detector is then reset and the readout cycle repeats for the next integration. The 4 readouts at the beginning are averaged together, the 4 readouts at the end are averaged together, and then the difference of the 2 averages is computed to form a final countrate image for each integration. This approach to creating the countrate images is used both on-board and in the *calwebb_guider* pipeline when the raw data are processed on the ground.

The FITS file structure for FineGuide uncalibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	uint16	8 x 8 x 8 x nints
2	Pointing	BINTABLE	N/A	12 cols x nrows
3	FGS Centroid Packet	BINTABLE	N/A	17 cols x nrows

- SCI: 4-D data array containing the raw pixel values.
- Pointing: A table containing guide star position and jitter information. See *above* for details of the contents.
- FGS Centroid Packet: A table containing guide star centroiding information. See *above* for details of the contents.

The FITS file structure for FineGuide calibrated products is as follows:

HDU	EXTNAME	HDU Type	Data Type	Dimensions
0	N/A	primary	N/A	N/A
1	SCI	IMAGE	float32	8 x 8 x nints
2	ERR	IMAGE	float32	8 x 8 x nints
3	DQ	IMAGE	uint32	8 x 8
4	POINTING	BINTABLE	N/A	12 cols x nrows
5	FGS CENTROID PACKET	BINTABLE	N/A	17 cols x nrows
6	ASDF	BINTABLE	N/A	variable

- SCI: 3-D data array containing the pixel values, in units of DN/s. Count rate images for each integration have been computed using the Fowler sampling scheme described above.
- ERR: 3-D data array containing uncertainty estimates for each pixel.

- DQ: 2-D data array containing DQ flags for each pixel.
- Pointing: A table containing guide star position and jitter information. See *above* for details of the contents.
- FGS Centroid Packet: A table containing guide star centroiding information. See *above* for details of the contents.
- ADSF: The data model meta data.

13.8 Migrating deprecated products

On rare occasion, the model schemas are changed in such a way as to break compatibility with data products produced by earlier versions of this package. When these older files are opened the software will report validation errors:

```
>>> from stdatamodels.jwst import datamodels
>>> datamodels.open("jw95115001001_02102_00001_nrs1_x1d.fits")
...
ValueError: Column names don't match schema...
```

In some cases it will be possible to update the file to the new format using the `migrate_data` tool included with this package:

```
$ migrate_data jw95115001001_02102_00001_nrs1_x1d.fits --in-place
```

It can also be run on multiple files:

```
$ migrate_data *_x1d.fits --in-place
```

Or configured to write updated files to a separate output directory:

```
$ migrate_data *_x1d.fits --output-dir some/other/directory
```

CHAPTER

FOURTEEN

ERROR PROPAGATION

14.1 Description

Steps in the various pipeline modules calculate variances due to different sources of noise or modify variances that were computed by previous steps. In some cases the variance arrays are only used internally within a given step. For several steps, these arrays must be propagated to subsequent steps in the pipeline. Anytime a step creates or updates variances, the total error (ERR) array values are always recomputed as the square root of the quadratic sum of all variances available at the time. Note that the ERR array values are always expressed as standard deviation (i.e. square root of the variance).

The table below is a summary of which steps create or update variance and error arrays, as well as which steps make use of these data. Details of how each step computes or uses these data are given in the subsequent sections below.

Step	Stage	Creates arrays	Updates arrays	Step uses
ramp_fitting	1	VAR_POISSON, VAR_RNOISE	ERR	None
gain_scale	1	None	ERR, VAR_POISSON, VAR_RNOISE	None
flat_field	2	VAR_FLAT	ERR, VAR_POISSON, VAR_RNOISE	None
fringe	2	None	ERR	None
barshadow	2	None	ERR, VAR_POISSON, VAR_RNOISE, VAR_FLAT	None
pathloss	2	None	ERR, VAR_POISSON, VAR_RNOISE, VAR_FLAT	None
photom	2	None	ERR, VAR_POISSON, VAR_RNOISE, VAR_FLAT	None
outlier_detection	3	None	None	ERR
resample	3	None	None	VAR_RNOISE
wfs_combine	3	None	ERR	None

14.2 Stage 1 Pipelines

Stage 1 pipelines perform detector-level corrections and ramp fitting for individual exposures, for nearly all imaging and spectroscopic modes. Details of the pipelines can be found at *Stage 1 Pipelines*.

The Stage 1 pipeline steps that alter the ERR, VAR_POISSON, or VAR_RNOISE arrays of the science countrate data are discussed below. Any step not listed here does not alter or use the variance or error arrays in any way and simply propagates the information to the next step.

14.2.1 ramp_fitting

This step calculates and populates the VAR_POISSON and VAR_RNOISE arrays in the 'rate' and 'rateints' files, and updates the ERR array as the square root of the quadratic sum of the variances. VAR_POISSON and VAR_RNOISE represent the uncertainty in the computed slopes (per pixel) due to Poisson and read noise, respectively. The details of the calculations can be found at *ramp_fitting*.

14.2.2 gain_scale

The `gain_scale` step is applied after `ramp_fitting`, and applies to both the rate and rateints products. The gain correction is applied to the ERR, VAR_POISSON, and VAR_RNOISE arrays. The SCI and ERR arrays are multiplied by the gain correction factor, and the variance arrays are multiplied by the square of the gain correction factor. More details can be found at *gain_scale*.

14.3 Stage 2 Pipelines

Stage 2 pipelines perform additional instrument-level and observing-mode corrections and calibrations to produce fully calibrated exposures. There are two main Stage 2 pipelines: one for imaging *calwebb_image2* and one for spectroscopy *calwebb_spec2*. In these pipelines, the various steps that apply corrections and calibrations apply those same corrections/calibrations to all variance arrays and update the total ERR array.

14.3.1 flat_field

The SCI array of the exposure being processed is divided by the flat-field reference image. The VAR_FLAT array is created, computed from the science data and the flat-field reference file ERR array.

For all modes except GUIDER, the VAR_POISSON and VAR_RNOISE arrays are divided by the square of the flat. The science data ERR array is then updated to be the square root of the sum of the three variance arrays.

For the GUIDER mode, their are no VAR_POISSON and VAR_RNOISE arrays. The science data ERR array is updated to be the square root of the sum of the variance VAR_FLAT and the square of the incoming science ERR array.

For more details see *flat_field*.

14.3.2 fringe

For MIRI MRS (IFU) mode exposures, the SCI and ERR arrays in the science exposure are divided by the fringe reference image. For details of the fringe correction, see *fringe*.

14.3.3 barshadow

This step is applied only to NIRSpec MSA data for extended sources. Once the 2-D correction array for each slit has been computed, it is applied to the science (SCI), error (ERR), and variance (VAR_POISSON, VAR_RNOISE, and VAR_FLAT) arrays of the slit. The correction values are divided into the SCI and ERR arrays, and the square of the correction values are divided into the variance arrays. For details of the bar shadow correction, see *barshadow*.

14.3.4 pathloss

The `pathloss` step corrects NIRSpec and NIRISS SOSS data for various types of light losses. The correction factors are divided into the SCI and ERR arrays of the science data, and the square of the correction values are divided into the variance arrays. For details of this step, see *pathloss*.

14.3.5 photom

The calibration information for the `photom` step includes a scalar flux conversion constant, as well as optional arrays of wavelength and relative response (as a function of wavelength). The combination of the scalar conversion factor and any 2-D response values is applied to the science data, including the SCI and ERR arrays, as well as the variance (VAR_POISSON, VAR_RNOISE, and VAR_FLAT) arrays. The flux calibration values are multiplied into the science exposure SCI and ERR arrays, and the square of the calibration values is multiplied into all variance arrays. For details of the photom correction, see *photom*.

14.4 Stage 3 pipelines

Stage 3 pipelines perform operations that work with multiple exposures and in most cases produce some kind of combined product. The operations in these pipelines that either use or modify variance/error arrays that are propagated through the pipeline are `outlier_detection` and `wfs_combine`.

14.4.1 outlier_detection

The `outlier_detection` step is used in all Stage 3 pipelines. It uses the ERR array to make a local noise model, based on the readnoise and calibration errors of earlier steps in the pipeline. This step does not modify the ERR array or any of the VAR arrays.

14.4.2 resample/resample_spec

The `resample` and `resample_spec` steps make use of the VAR_RNOISE array to compute weights that are used when combining data with the `weight_type=ivm` option selected. The step also resamples all of the variance and error arrays, using the same output WCS frame as the science data.

14.4.3 wfs_combine

The `wfs_combine` step is only applied in the Stage 3 Wavefront Sensing and Control (calwebb_wfs-image3) pipeline for dithered pairs of WFS&C exposures. This step can modify variance/error arrays, but only if the optional "do_refine" parameter is set to True (the default is False). In this case the algorithm to refine image offsets will be used and the ERR array values will be altered on output, using a combination of the input image errors. See the step documentation at *wfs_combine* for more details.

CHAPTER

FIFTEEN

PACKAGE DOCUMENTATION

15.1 Package Index

15.1.1 Align PSF References

Description

Class
`jwst.coron.AlignRefsStep`

Alias
align_refs

The `align_refs` step is one of the coronagraphic-specific steps in the `coron` sub-package that is part of Stage 3 *calwebb_coron3* processing. It computes offsets between science target images and reference PSF images, and shifts the PSF images into alignment. This is performed on a per-integration basis for both the science target data and the reference PSF data. Each integration contained in the stacked PSF data (the result of the *stack_refs*) step is aligned to each integration within a given science target exposure. This results in a new product for each science target exposure that contains a stack of individual PSF images that have been aligned to each integration in the science target exposure.

Shifts between each PSF and target image are computed using the `scipy.optimize.leastsq` function. A 2D mask, supplied via a PSFMASK reference file, is used to indicate pixels to ignore when performing the minimization in the `leastsq` routine. The mask acts as a weighting function in performing the fit. Alignment of a PSF image is performed using the `scipy.ndimage.fourier_shift` function and the computed sub-pixel offsets.

Arguments

The `align_refs` step has two optional arguments:

`--median_box_length` (integer, default=3)
The box size to use when replacing bad pixels with the median in a surrounding box.

`--bad_bits` (string, default="DO_NOT_USE")
The DQ bit values from the input image DQ arrays that should be considered bad and replaced with the median in a surrounding box. For example, setting to `"OUTLIER, SATURATED"` (or equivalently `"16, 2"` or `"18"`) will treat all pixels flagged as OUTLIER or SATURATED as bad, while setting to `""` or `None` will treat all pixels as good and omit any bad pixel replacement.

Inputs

The `align_refs` step takes 2 inputs: a science target exposure containing a 3D stack of calibrated per-integration images and a "_psfstack" product containing a 3D stack of reference PSF images. If the target or PSF images have any of the data quality flags set to those specified by the `bad_bits` argument, these pixels are replaced with the median value of a region around the flagged data. The size of the box region to use for the replacement can also be specified. These corrected images are used in the *align_refs* step and passed along for subsequent processing.

3D calibrated images

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_calints

One of the science target exposures specified in the ASN file used as input to the *calwebb_coron3* pipeline. This should be a "_calints" product from the *calwebb_image2* pipeline and contains a 3D stack of per-integration images.

3D stacked PSF images

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_psfstack

A "_psfstack" product created by the *stack_refs* step, which contains the collection of all PSF images to be used, in the form of a 3D image stack.

Outputs

4D aligned PSF images

Data model
`QuadModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.QuadModel.html#jwst.datamodels.QuadMod

File suffix
_psfalign

The output is a 4D data model, where the 3rd axis has length equal to the total number of reference PSF images in the input PSF stack and the 4th axis has length equal to the number of integrations in the input science target product (ncols x nrows x npsfs x nints). Image[n,m] in the 4D data is the n th PSF image aligned to the m th science target integration. The file name is exposure-based, using the input science target exposure name as the root, with the addition of the association candidate ID and the "_psfalign" product type suffix, e.g. "jw8607342001_02102_00001_nrcb3_a3001_psfalign.fits."

Reference Files

The `align_refs` step uses a PSFMASK reference file.

PSFMASK Reference File

REFTYPE
PSFMASK

Data model
`PsfMaskModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.PsfMaskModel.html#jwst.datamodels.Psf

The PSFMASK reference file contains a 2-D mask that's used as a weight function when computing shifts between images.

Reference Selection Keywords for PSFMASK

CRDS selects appropriate PSFMASK references based on the following keywords. PSFMASK is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, FILTER, CORONMSK, SUBARRAY, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, FILTER, CORONMSK, SUBARRAY, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for PSFMASK

In addition to the standard reference file keywords listed above, the following keywords are *required* in PSFMASK reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for PSFMASK*):

Keyword	Data Model Name
FILTER	model.meta.instrument.filter
CORONMSK	model.meta.instrument.coronagraph
SUBARRAY	model.meta.subarray.name

Reference File Format

PSFMASK reference files are FITS format, with 1 IMAGE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float

The values in the `SCI` array give the mask values to be applied to the images when computing relative shifts. The mask acts as a weighting function when performing Fourier fits. The values range from zero (full weighting) to one (pixel completely masked out).

jwst.coron.align_refs_step Module

Replace bad pixels and align psf image with target image.

Classes

AlignRefsStep([name, parent, config_file, ...])	AlignRefsStep: Align coronagraphic PSF images with science target images.

AlignRefsStep

class jwst.coron.align_refs_step.**AlignRefsStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

AlignRefsStep: Align coronagraphic PSF images with science target images.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.

- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(target, psf)	This is where real work happens.

Attributes Documentation

class_alias = 'align_refs'

reference_file_types = ['psfmask']

spec

```
median_box_length = integer(default=3,min=0) # box size for the median filter
bad_bits = string(default="DO_NOT_USE") # the DQ bit values of bad pixels
```

Methods Documentation

process(*target*, *psf*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.2 AMI Analyze

Description

Class
`jwst.ami.AmiAnalyzeStep`

Alias
ami_analyze

The `ami_analyze` step is one of the AMI-specific steps in the `ami` sub-package that is part of Stage 3 *calwebb_ami3* processing. It applies the Lacour-Greenbaum (LG) image plane modeling algorithm to a NIRISS AMI image. The routine computes a number of parameters, including a model fit (and residuals) to the image, fringe amplitudes and phases, and closure phases and amplitudes.

The JWST AMI observing template allows for exposures to be obtained using either full-frame (SUBARRAY="FULL") or subarray (SUBARRAY="SUB80") readouts. When processing a full-frame exposure, the `ami_analyze` step extracts and processes a region from the image corresponding to the size and location of the SUB80 subarray, in order to reduce execution time.

Arguments

The `ami_analyze` step has four optional arguments:

–oversample
The oversampling factor to be used in the model fit (default=3).

–rotation
Initial guess for the rotation of the PSF in the input image, in units of degrees (default=0.0).

–psf_offset
List of PSF offset values to use when creating the model array (default='0.0 0.0').

–rotation_search
List of start, stop, and step values that define the list of rotation search values. The default setting of '-3 3 1' results in search values of [-3, -2, -1, 0, 1, 2, 3].

Inputs

2D calibrated image

Data model
ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_cal

The ami_analyze step takes a single calibrated image as input, which should be the "_cal" product resulting from *calwebb_image2* processing. Multiple exposures can be processed via use of an ASN file that is used as input to the *calwebb_ami3* pipeline. The ami_analyze step itself does not accept an ASN as input.

Outputs

LG model parameters

Data model
AmiLgModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_ami

The ami_analyze step produces a single output file, containing the following list of extensions:

1) FIT: a 2D image of the fitted model
2) RESID: a 2D image of the fit residuals
3) CLOSURE_AMP: table of closure amplitudes
4) CLOSURE_PHA: table of closure phases
5) FRINGE_AMP: table of fringe amplitudes
6) FRINGE_PHA: table of fringe phases
7) PUPIL_PHA: table of pupil phases
8) SOLNS: table of fringe coefficients

The output file name syntax is exposure-based, using the input file name as the root, with the addition of the association candidate ID and the "_ami" product type suffix, e.g. "jw87600027001_02101_00002_nis_a3001_ami.fits."

Reference Files

The ami_analyze step uses a THROUGHPUT reference file.

THROUGHPUT Reference File

REFTYPE
THROUGHPUT

Data model
`ThroughputModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ThroughputModel.html#jwst.datamod

The THROUGHPUT reference file contains throughput data for the filter used in the AMI image.

Reference Selection Keywords for THROUGHPUT

CRDS selects appropriate THROUGHPUT references based on the following keywords. THROUGHPUT is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
NIRISS	INSTRUME, FILTER, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for THROUGHPUT

In addition to the standard reference file keywords listed above, the following keywords are *required* in THROUGHPUT reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for THROUGHPUT*):

Keyword	Data Model Name
FILTER	model.meta.instrument.filter

Reference File Format

THROUGHPUT reference files are FITS files with one BINTABLE extension. The FITS primary data array is assumed to be empty. The format of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
THROUGHPUT	BINTABLE	2	TFIELDS = 2	N/A

The table extension contains two columns, giving wavelength and throughput values for a particular filter:

Column name	Data type	Units
wavelength	float	Angstroms
throughput	float	(unitless)

AMI unit tests

There are unit tests for AMI Analyze and AMI interface.

test_ami_interface

- Make sure ami_analyze fails if input is an input file of type _calints
- Make sure ami_analyze fails if input is CubeModel for _calints
- Make sure that ami_analyze fails if no throughput reffile is available

test_ami_analyze

utils module tests:

For the module *utils* we have several tests that compare the calculated value with a known value. The tests are:

- Test of rebin() and krebin()
- Test of quadratic
- Test of findmax
- Test of makeA
- Test of fringes2pistons
- Test of rcrosscorrelate()
- Test of crosscorrelate()

leastsqnrm module tests:

- **Test of rotatevectors()**
  Positive x decreases under slight rotation, and positive y increases under slight rotation.
- **Test of flip()**
  Change sign of 2nd coordinate of holes.
- **Test of mas2rad()**
  Convert angle in milli arc-sec to radians.
- **Test of rad2mas()**
  Convert input angle in radians to milli arc sec.
- **Test of sin2deltapistons()**
  Each baseline has one sine and one cosine fringe with a coefficient that depends on the piston difference between the two holes that make the baseline. For a 7-hole mask there are 21 baselines and therefore there are 42 sine and cosine terms that contribute to the fringe model. This function calculates the sine of this piston difference.
- **Test of cos2deltapistons()**
  Each baseline has one sine and one cosine fringe with a coefficient that depends on the piston difference between the two holes that make the baseline. For a 7-hole mask there are 21 baselines and therefore there are 42 sine and cosine terms that contribute to the fringe model. This function calculate the cosine of this piston difference.
- **Test of replacenan()**
  Replace singularities encountered in the analytical hexagon Fourier transform with the analytically derived limits. (pi/4)
- **Test of hexpb()**
  Calculate the primary beam for hexagonal holes.
- **Test of model_array**
  Create a model using the specified wavelength.
- **Test of ffc**
  Calculate cosine terms of analytic model.
- **Test of ffs**
  Calculate sine terms of analytic model.
- **Test of return_CAs**
  Calculate the closure amplitudes.
- **Test of closurephase**
  Calculate closure phases between each pair of holes.
- **Test of redundant_cps**
  Calculate closure phases for each set of 3 holes.
- **Test of populate_symmamparray**
  Populate the symmetric fringe amplitude array.
- **Test of populate_antisymmphasearray**
  Populate the antisymmetric fringe phase array.
- **Test of tan2visibilities**
  From the solution to the fit, calculate the fringe amplitude and phase.
- **Test of multiplyenv**
  Multiply the envelope by each fringe 'image'.

hexee module tests:

- **Test of g_eeAG()**
  Calculate the Fourier transform of one half of a hexagon that is bisected from one corner to its diametrically opposite corner.
- **Test of glimit()**
  Calculate the analytic limit of the Fourier transform of one half of the hexagon along eta=0.

analyticnrm2 module tests:

- Test of PSF()
- Test of ASFhex() in the analyticnrm2 module FOR HEX
- Test of interf()
- Test of phasor()

webb_psf module test:

- **Test of PSF()**
  Create a Throughput datamodel, having a dummy filter bandpass data that peaks at 1.0 at the center and decreases in the wings.

jwst.ami.ami_analyze_step Module

Classes

AmiAnalyzeStep([name, parent, config_file, ...])	Performs analysis of an AMI mode exposure by applying the LG algorithm.

AmiAnalyzeStep

class jwst.ami.ami_analyze_step.**AmiAnalyzeStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Performs analysis of an AMI mode exposure by applying the LG algorithm.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.

- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	Performs analysis of an AMI mode exposure by applying the LG algorithm.

Attributes Documentation

class_alias = 'ami_analyze'

reference_file_types = ['throughput']

spec

```
oversample = integer(default=3, min=1)  # Oversampling factor
rotation = float(default=0.0)           # Rotation initial guess [deg]
psf_offset = string(default='0.0 0.0') # Psf offset values to use to create the␣
↪model array
rotation_search = string(default='-3 3 1') # Rotation search parameters: start,␣
↪stop, step
```

Methods Documentation

process(*input*)

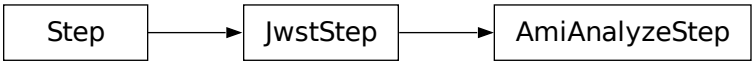
Performs analysis of an AMI mode exposure by applying the LG algorithm.

Parameters
input (*string*) – input file name

Returns
result – AMI image to which the LG fringe detection has been applied

Return type
AmiLgModel object

Class Inheritance Diagram



15.1.3 AMI Average

Description

Class
`jwst.ami.AmiAverageStep`

Alias
ami_average

The `ami_average` step is one of the AMI-specific steps in the `ami` sub-package and is part of Stage 3 *calwebb_ami3* processing. It averages the results of LG processing from the *ami_analyze* step for multiple exposures of a given target. It computes a simple average for all 8 components of the "ami" product files from all input exposures.

For a given association of exposures, the "ami" products created by the `ami_analyze` step may have `fit_image` and `resid_image` images that vary in size from one exposure to another. If this is the case, the smallest image size of all the input products is used for the averaged product and the averaged `fit_image` and `resid_image` images are created by trimming extra rows/columns from the edges of images that are larger.

Arguments

The `ami_average` step does not have any step-specific arguments.

Inputs

LG model parameters

Data model
`AmiLgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_ami

The only input to the `ami_average` step is a list of one or more "ami" files to be processed. These should be output files from the *ami_analyze* step. The input to the step must be in the form of a list of "ami" **file names**. Passing data models or ASN files is not supported at this time. Use the *calwebb_ami3* pipeline to conveniently process multiple inputs.

Outputs

Average LG model parameters

Data model
AmiLgModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_amiavg or _psf-amiavg

The ami_average step produces a single output file, having the same format as the input files, where the data for the 8 file components are the averages from the list of input files. If the inputs in the ASN file are designated as "science", the output product type will be "_amiavg", whereas if the inputs are designated as "psf", the output product type will be "_psf-amiavg." The output file name syntax is source-based, using the product name specified in the input ASN file, e.g. "jw87600-a3001_t001_niriss_f480m-nrm_amiavg.fits."

Reference Files

The ami_average step does not use any reference files.

jwst.ami.ami_average_step Module

Classes

AmiAverageStep([name, parent, config_file, ...])	AmiAverageStep: Averages LG results for multiple NIRISS AMI mode exposures

AmiAverageStep

class jwst.ami.ami_average_step.**AmiAverageStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

AmiAverageStep: Averages LG results for multiple NIRISS AMI mode exposures

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

flatten_input(input_items)	Remove any nested list/tuple structure and return generator to provide iterable simple list with no nested structure.
process(*input_list)	Averages the results of LG analysis for a set of multiple NIRISS AMI mode exposures.

Attributes Documentation

class_alias = 'ami_average'

spec

Methods Documentation

flatten_input(*input_items*)

Remove any nested list/tuple structure and return generator to provide iterable simple list with no nested structure.

process(**input_list*)

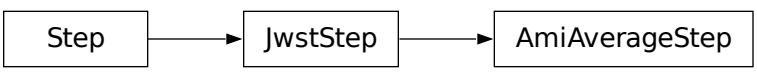
Averages the results of LG analysis for a set of multiple NIRISS AMI mode exposures.

Parameters
input_list (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – input file names

Returns
result – Averaged AMI data model

Return type
AmiLgModel object

Class Inheritance Diagram



15.1.4 AMI Normalize

Description

Class
`jwst.ami.AmiNormalizeStep`

Alias
ami_normalize

The `ami_normalize` step is one of the AMI-specific steps in the `ami` sub-package and is used in Stage 3 *calwebb_ami3* processing. It provides normalization of LG processing results for a science target using LG results of a reference PSF target. The algorithm subtracts the PSF target closure phases from the science target closure phases and divides the science target fringe amplitudes by the PSF target fringe amplitudes.

Arguments

The `ami_normalize` step does not have any step-specific arguments.

Inputs

LG model parameters

Data model
`AmiLgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_amiavg and _psf-amiavg

The `ami_normalize` step takes two inputs: the first is the LG results for a science target and the second is the LG results for the PSF target. These should be the "_amiavg" and "_psf-amiavg" products resulting from the *ami_average* step. The inputs can be in the form of file names or `AmiLgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLgModel) data models.

Outputs

Normalized LG model parameters

Data model
`AmiLgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_aminorm

The output is a new LG product for the science target in which the closure phases and fringe amplitudes have been normalized using the PSF target closure phases and fringe amplitudes. The remaining components of the science target data model are left unchanged. The output file name syntax is source-based, using the product name specified in the input ASN file and having a product type of "_aminorm", e.g. "jw87600-a3001_t001_niriss_f480m-nrm_aminorm.fits."

Reference Files

The ami_normalize step does not use any reference files.

jwst.ami.ami_normalize_step Module

Classes

AmiNormalizeStep([name, parent, ...])	AmiNormalizeStep: Normalize target LG results using reference LG results

AmiNormalizeStep

class jwst.ami.ami_normalize_step.**AmiNormalizeStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

AmiNormalizeStep: Normalize target LG results using reference LG results

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(target, reference)	Normalizes the LG results for a science target, using the LG results for a reference target.

Attributes Documentation

class_alias = 'ami_normalize'

spec

Methods Documentation

process(*target*, *reference*)

Normalizes the LG results for a science target, using the LG results for a reference target.

Parameters

- **target** (*string or model*) – target input
- **reference** (*string or model*) – reference input

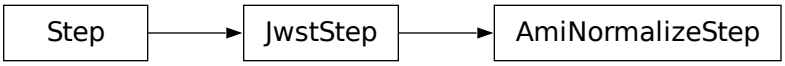
Returns

result – AMI data model that's been normalized

Return type

AmiLgModel object

Class Inheritance Diagram



15.1.5 Assign Moving Target WCS

Description

Class

jwst.assign_mtwcs.AssignMTWcsStep

Alias

assign_mtwcs

The `jwst.assign_mtwcs` step modifies the WCS output frame in each exposure of a Moving Target (MT) observation association, such that the WCS is centered at the average location of the target within the whole association. This results in proper alignment of multiple exposures, which takes place downstream in the calibration pipeline, in the target frame, rather than the background sky frame.

A moving target will naturally be located at different sky coordinates (RA, Dec) across multiple exposures within an MT observation. When multiple images or spectra get combined during Stage 3 processing, the relative alignment of the images/spectra is based on the sky coordinates of each exposure. In the case of moving targets, where the RA/Dec of the target is changing between exposures, the normal alignment process would result in the target being at different image coordinates and hence coming out either smeared (for slowly moving targets) or at multiple locations within the combined data. This step modifies the WCS of each exposure to recenter it at a common RA/Dec for the target, so that subsequent image alignment and combination has the target properly aligned.

The step is executed at the beginning of the *calwebb_image3* and *calwebb_spec3* pipelines, so that all subsequent steps that rely on WCS information use the frame centered on the target.

This step depends on keywords that are unique to MT exposures, as shown in the following table.

FITS Keyword	Data model attribute	Type (Value)	Description
TARGTYPE	meta.target.type	string (moving)	Type of target
MT_RA MT_DEC	meta.wcsinfo.mt_ra meta.wcsinfo.mt_dec	number number	Target RA and Dec at mid-point of exposure [deg]
MT_AVRA MT_AVDEC	meta.wcsinfo.mt_avra meta.wcsinfo.mt_avdec	number number	Target RA and Dec averaged between exposures [deg]

A "TARGTYPE" value of "moving" is used to identify exposures as containing a moving target. The keywords "MT_RA" and "MT_DEC" are populated in the uncalibrated (*uncal*) product for each exposure and give the position of the target at the mid-point of each exposure. The `assign_mtwcs` step computes the average of the "MT_RA" and "MT_DEC" values across all expsoures in an association and stores the result in the "MT_AVRA" and "MT_AVDEC" keywords of each exposure.

In addition to populating the "MT_AVRA" and "MT_AVDEC" keywords, this step adds another transform to the original WCS in each exposure that results in the WCS frame being centered at "MT_AVRA" and "MT_AVDEC". The transform of the original WCS associated with the science aperture pointing (i.e. without the additional MT correction) can be accessed by executing:

```
sci_transform = model.meta.wcs.get_transform('detector', 'world')
```

jwst.assign_mtwcs Package

Classes

AssignMTWcsStep([name, parent, config_file, ...])	AssignMTWcsStep: Create a gWCS object for a moving target.

AssignMTWcsStep

class jwst.assign_mtwcs.**AssignMTWcsStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

AssignMTWcsStep: Create a gWCS object for a moving target.

Parameters
input (*Association*) – A JWST association file.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'assign_mtwcs'

spec

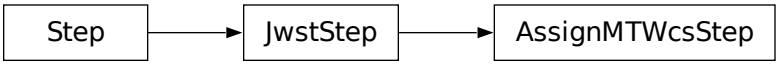
```
suffix = string(default='assign_mtwcs')     # Default suffix for output files
output_use_model = boolean(default=True)    # When saving use `DataModel.meta.
↪filename`
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.6 Assign WCS

Description

Class

jwst.assign_wcs.AssignWcsStep

Alias

assign_wcs

`jwst.assign_wcs` is run in the beginning of the level 2B JWST pipeline. It associates a WCS object with each science exposure. The WCS object transforms positions in the detector frame to positions in a world coordinate frame - ICRS and wavelength. In general there may be intermediate coordinate frames depending on the instrument. The WCS is saved in the ASDF extension of the FITS file. It can be accessed as an attribute of the meta object when the fits file is opened as a data model.

The forward direction of the transforms is from detector to world coordinates and the input positions are 0-based.

`jwst.assign_wcs` expects to find the basic WCS keywords in the SCI header. Distortion and spectral models are stored in reference files in the ASDF (http://asdf-standard.readthedocs.org/en/latest/) format.

For each observing mode, determined by the value of `EXP_TYPE` in the science header, assign_wcs retrieves reference files from CRDS and creates a pipeline of transforms from input frame `detector` to a frame `v2v3`. This part of the WCS pipeline may include intermediate coordinate frames. The basic WCS keywords are used to create the transform from frame `v2v3` to frame `world`.

For image display with software like DS9 that relies on specific WCS information, a SIP-based approximation to the WCS is fit. The results are FITS keywords stored in `model.meta.wcsinfo`. This is not an exact fit, but is accurate to ~0.25 pixel and is sufficient for display purposes. This step, which occurs for imaging modes early, is performed by default but can be switched off, and parameters controlling the fit can also be adjusted.

`jwst.assign_wcs` is based on gwcs (https://gwcs.readthedocs.io/en/latest/) and uses asdf (http://asdf.readthedocs.io/en/latest/).

Basic WCS keywords and the transform from `v2v3` to `world`

All JWST instruments use the following FITS header keywords to define the transform from `v2v3` to `world`:

`RA_REF`, `DEC_REF` - a fiducial point on the sky, ICRS, [deg]

`V2_REF`, `V3_REF` - a point in the V2V3 system which maps to `RA_REF`, `DEC_REF`, [arcsec]

`ROLL_REF` - local roll angle associated with each aperture, [deg]

`RADESYS` - standard coordinate system [ICRS]

These quantities are used to create a 3D Euler angle rotation between the V2V3 spherical system, associated with the telescope, and a standard celestial system.

For spectroscopic data, `jwst.assign_wcs` populates keyword `DISPAXIS` with an integer value that indicates whether the dispersion direction is oriented more nearly along the horizontal (DISPAXIS = 1) or vertical (DISPAXIS = 2) direction.

Using the WCS interactively

Once a FITS file is opened as a `DataModel` the WCS can be accessed as an attribute of the meta object. Calling it as a function with detector positions as inputs returns the corresponding world coordinates. Using MIRI LRS fixed slit as an example:

```
>>> from stdatamodels.jwst.datamodels import ImageModel
>>> exp = ImageModel('miri_fixedslit_assign_wcs.fits')
>>> ra, dec, lam = exp.meta.wcs(x, y)
>>> print(ra, dec, lam)
    (329.97260532549336, 372.0242999250267, 5.4176100046836675)
```

The WFSS modes for NIRCam and NIRISS have a slightly different calling structure, in addition to the (x, y) coordinate, they need to know other information about the spectrum or source object. In the JWST backward direction (going from the sky to the detector) the WCS model also looks for the wavelength and order and returns the (x,y) location of that wavelength+order on the dispersed image and the original source pixel location, as entered, along with the order that was specified:

```
>>> from stdatamodels.jwst.datamodels import ImageModel
>>> exp = ImageModel('nircam_wfss_assign_wcs.fits')
>>> x, y, x0, y0, order = exp.meta.wcs(x0, y0, wavelength, order)
>>> print(x0, y0, wavelength, order)
    (365.523884327, 11.6539963919, 2.557881113, 2)
>>> print(x, y, x0, y0, order)
    (1539.5898464615102, 11.6539963919, 365.523884327, 11.6539963919, 2)
```

The WCS provides access to intermediate coordinate frames and transforms between any two frames in the WCS pipeline in forward or backward direction. For example, for a NIRSpec fixed slits exposure, which has been through the extract_2d step:

```
>>> exp = datamodels.MultiSlitModel('nrs1_fixed_assign_wcs_extract_2d.fits')
>>> exp.slits[0].meta.wcs.available_frames
    ['detector', 'sca', 'bgwa', 'slit_frame', 'msa_frame', 'ote', 'v2v3', 'world']
>>> msa2detector = exp.slits[0].meta.wcs.get_transform('msa_frame', 'detector')
>>> msa2detector(0, 0, 2*10**-6)
    (5042.064255529629, 1119.8937888372516)
```

For each exposure, assign_wcs uses reference files and WCS header keywords to create the WCS object. What reference files are retrieved from CRDS is determined based on EXP_TYPE and other keywords in the science file header.

The assign_wcs step can accept the single slope image that is the result of averaging over all integrations or a 3D cube of integrations in the case of TSO exposures.

WCS of slitless grism exposures

The WCS forward transforms for slitless grism exposures (`NIS_WFSS`, `NRC_WFSS`, `NRC_TSGRISM`) take as input the `x, y` coordinates on the dispersed image, the `x0, y0` coordinate of the center of the object in the direct image and `spectral order`. They return the `x0, y0` coordinate of the center of the object in the direct image, `wavelength` and `spectral order`.

For NIRISS WFSS data the reference files contain a reference value for the filter wheel position angle. The trace is rotated about an angle which is the difference between the reference and actual angles.

For WFSS modes (`NIS_WFSS`, `NRC_WFSS`), an approximation of the GWCS object associated with a direct image with the same instrument configuration as the grism image is saved as FITS WCS in the headers of grism images.

Corrections Due to Spacecraft Motion

The WCS transforms contain two corrections due to motion of the observatory.

Absolute velocity aberration is calculated onboard when acquiring the guide star, but differential velocity aberration effects are calculated during the `assign_wcs` step. This introduces corrections in the conversion from sky coordinates to observatory V2/V3 coordinates, and is stored in the WCS under the `v2v3vacorr` frame.

For spectroscopic data, a relativistic Doppler correction is applied to all wavelengths to place observations into the barycentric reference frame. This correction factor is applied to the WCS wavelength solution created during the `assign_wcs` step, such that extracted spectral products will have wavelength arrays in the barycentric frame.

Step Arguments

The `assign_wcs` step has the following optional arguments to control the behavior of the processing.

`--sip_approx` (boolean, default=True)
A flag to enable the computation of a SIP approximation for imaging modes.

`--sip_degree` (integer, max=6, default=None)
Polynomial degree for the forward SIP fit. "None" uses the best fit.

`--sip_max_pix_error` (float, default=0.1)
Maximum error for the SIP forward fit, in units of pixels. Ignored if `sip_degree` is set to an explicit value.

`--sip_inv_degree` (integer, max=6, default=None)
Polynomial degree for the inverse SIP fit. "None" uses the best fit.

`--sip_max_inv_pix_error` (float, default=0.1)
Maximum error for the SIP inverse fit, in units of pixels. Ignored if `sip_inv_degree` is set to an explicit value.

--sip_npoints (integer, default=12)
Number of points for the SIP fit.

--slit_y_low (float, default=-0.55)
Lower edge of a NIRSpec slit.

--slit_y_high (float, default=0.55)
Upper edge of a NIRSpec slit.

Reference Files

WCS Reference files are in the Advanced Scientific Data Format (ASDF). The best way to create the file is to programmatically create the model and then save it to a file. A tutorial on creating reference files in ASDF format is available at:

https://github.com/spacetelescope/jwreftools/blob/master/docs/notebooks/referece_files_asdf.ipynb

Transforms are 0-based. The forward direction is from detector to sky.

Reference file types used by `assign_wcs`

REFTYPE	Description	Instruments
CAMERA	NIRSpec Camera model	NIRSpec
COLLIMATOR	NIRSpec Collimator Model	NIRSpec
DISPERSER	Disperser parameters	NIRSpec
DISTORTION	Spatial distortion model	FGS, MIRI, NIRCam, NIRISS
FILTEROFFSET	MIRI Imager filter offsets	MIRI, NIRCAM, NIRISS
FORE	Transform through the NIRSpec FORE optics	NIRSpec
FPA	Transform in the NIRSpec FPA plane	NIRSpec
IFUFORE	Transform from the IFU slicer to the IFU entrance	NIRSpec
IFUPOST	Transform from the IFU slicer to the back of the IFU	NIRSpec
IFUSLICER	IFU Slicer geometric description	NIRSpec
MSA	Transform in the NIRSpec MSA plane	NIRSpec
OTE	Transform through the Optical Telescope Element	NIRSpec
SPECWCS	Wavelength calibration models	MIRI, NIRCam, NIRISS
REGIONS	Stores location of the regions on the detector	MIRI
WAVELENGTH-RANGE	Typical wavelength ranges	MIRI, NIRCam, NIRISS, NIRSpec

How To Create Reference files in ASDF format

All WCS reference files are in ASDF (http://asdf-standard.readthedocs.org/en/latest/) format. ASDF is a human-readable, hierarchical metadata structure, made up of basic dynamic data types such as strings, numbers, lists and mappings. Data is saved as binary arrays. It is primarily intended as an interchange format for delivering products from instruments to scientists or between scientists. It's based on YAML and JSON schema and as such provides automatic structure and metadata validation.

While it is possible to write or edit an ASDF file in a text editor, or to use the ASDF interface, the best way to create reference files is using the datamodels in the jwst pipeline jwst.datamodels (http://jwst-pipeline.readthedocs.io/en/latest/jwst/datamodels/index.html#classes) and astropy.modeling (http://astropy.readthedocs.io/en/latest/modeling/index.html) .

There are two steps in this process:

- create a transform using the simple models and the rules to combine them
- save the transform to an ASDF file (this automatically validates it)

The rest of this document provides a brief description and examples of models in astropy.modeling (http://astropy.readthedocs.org/en/latest/modeling/index.html) which are most relevant to WCS and examples of creating WCS reference files.

Create a transform

astropy.modeling (http://astropy.readthedocs.org/en/latest/modeling/index.html) is a framework for representing, evaluating and fitting models. All available models can be imported from the `models` module.

```
>>> from astropy.modeling import models as astmodels
```

If necessary all fitters can be imported through the `fitting` module.

```
>>> from astropy.modeling import fitting
```

Many analytical models are already implemented and it is easy to implement new ones. Models are initialized with their parameter values. They are evaluated by passing the inputs directly, similar to the way functions are called. For example,

```
>>> poly_x = astmodels.Polynomial2D(degree=2, c0_0=.2, c1_0=.11, c2_0=2.3, c0_1=.43, c0_
↪2=.1, c1_1=.5)
>>> poly_x(1, 1)
    3.639999
```

Models have their analytical inverse defined if it exists and accessible through the `inverse` property. An inverse model can also be (re)defined by assigning to the `inverse` property.

```
>>> rotation = astmodels.Rotation2D(angle=23.4)
>>> rotation.inverse
    <Rotation2D(angle=-23.4)>
>>> poly_x.inverse = astmodels.Polynomial2D(degree=3, **coeffs)
```

astropy.modeling also provides the means to combine models in various ways.

Model concatenation uses the & operator. Models are evaluated on independent inputs and results are concatenated. The total number of inputs must be equal to the sum of the number of inputs of all models.

```
>>> shift_x = astmodels.Shift(-34.2)
>>> shift_y = astmodels.Shift(-120)
>>> model = shift_x & shift_y
>>> model(1, 1)
    (-33.2, -119.0)
```

Model composition uses the | operator. The output of one model is passed as input to the next one, so the number of outputs of one model must be equal to the number of inputs to the next one.

```
>>> model = poly_x | shift_x | astmodels.Scale(-2.3)
>>> model = shift_x & shift_y | poly_x
```

Two models, `Mapping` and `Identity`, are useful for axes manipulation - dropping or creating axes, or switching the order of the inputs.

`Mapping` takes a tuple of integers and an optional number of inputs. The tuple represents indices into the inputs. For example, to represent a 2D Polynomial distortion in `x` and `y`, preceded by a shift in both axes:

```
>>> poly_y = astmodels.Polynomial2D(degree=2, c0_0=.2, c1_0=1.1, c2_0=.023, c0_1=3, c0_
↪2=.01, c1_1=2.2)
>>> model = shift_x & shift_y | astmodels.Mapping((0, 1, 0, 1)) | poly_x & poly_y
>>> model(1, 1)
    (5872.03, 8465.401)
```

`Identity` takes an integer which represents the number of inputs to be passed unchanged. This can be useful when one of the inputs does not need more processing. As an example, two spatial (V2V3) and one spectral (wavelength) inputs are passed to a composite model which transforms the spatial coordinates to celestial coordinates and needs to pass the wavelength unchanged.

```
>>> tan = astmodels.Pix2Sky_TAN()
>>> model = tan & astmodels.Identity(1)
>>> model(0.2, 0.3, 10**-6)
   (146.30993247402023, 89.63944963170002, 1e-06)
```

Arithmetic Operators can be used to combine models. In this case each model is evaluated with all inputs and the operator is applied to the results, e.g. `model = m1 + m2 * m3 - m4/m5**m6`

```
>>> model = shift_x + shift_y
>>> model(1)
    -152.2
```

Create the reference file

The DistortionModel in jwst.datamodels is used as an example of how to create a reference file. Similarly data models should be used to create other types of reference files as this process provides validation of the file structure.

```
>>> from stdatamodels.jwst.datamodels import DistortionModel
>>> dist = DistortionModel(model=model, strict_validation=True)
>>> dist.meta.description = "New distortion model"
>>> dist.meta.author = "INS team"
>>> dist.meta.useafter = "2012/01/21"
>>> dist.meta.instrument.name = "MIRI"
>>> dist .meta.instrument.detector = "MIRIMAGE"
>>> dist.meta.pedigree = "GROUND"
>>> dist.meta.reftype = "distortion"
>>> dist.meta.input_units = "pixel"
>>> dist.meta.output_units = "arcsec"
>>> dist.validate()
>>> dist.save("new_distortion.asdf")
'new_distortion.asdf'
```

Save a transform to an ASDF file

asdf (http://asdf.readthedocs.io/en/latest/) is used to read and write reference files in ASDF (http://asdf-standard.readthedocs.org/en/latest/) format. Once the model has been created using the rules in the above section, it needs to be assigned to the ASDF tree.

```
>>> import asdf
>>> from asdf import AsdfFile
>>> f = AsdfFile()
>>> f.tree['model'] = model
>>> f.write_to('reffile.asdf')
```

The `write_to` command validates the file and writes it to disk. It will catch any errors due to inconsistent inputs/outputs or invalid parameters.

To test the file, it can be read in again using the `asdf.open()` function:

```
>>> with asdf.open('reffile.asdf') as ff:
...     model = ff.tree['model']
...     model(1)
    -152.2
```

WCS reference file information per EXP_TYPE

FGS_IMAGE, FGS_FOCUS, FGS_SKYFLAT, FGS_INTFLAT

reftypes: *distortion*
WCS pipeline coordinate frames: detector, v2v3, world
Implements: reference file provided by NIRISS team

MIR_IMAGE, MIR_TACQ, MIR_LYOT, MIR4QPM, MIR_CORONCAL

reftypes: *distortion*, *filteroffset*
WCS pipeline coordinate frames: detector, v2v3, world
Implements: CDP6 reference data delivery,
MIRI-TN-00070-ATC_Imager_distortion_CDP_Iss5.pdf

MIR_LRS-FIXEDSLIT, MIR_LRS-SLITLESS

reftypes: *specwcs*, *distortion*
WCS pipeline coordinate frames: detector, v2v3, world
Implements: CDP4 reference data delivery,
MIRI-TR-10020-MPI-Calibration-Data-Description_LRSPSFDistWave_v4.0.pdf

MIR_MRS

reftypes: *distortion*, *specwcs*, *v2v3*, *wavelengthrange*, *regions*
WCS pipeline coordinate frames: detector, miri_focal, xyan, v2v3, world
Implements: CDP4 reference data delivery,
MIRI-TN-00001-ETH_Iss1-3_Calibrationproduct_MRS_d2c.pdf

NRC_IMAGE, NRC_TSIMAGE, NRC_FOCUS, NRC_TACONFIRM, NRC_TACQ

reftypes: *distortion*, *filteroffset*
WCS pipeline coordinate frames: detector, v2v3, world
Implements: Distortion file created from TEL team data.

NRC_WFSS, NRC_TSGRISM

reftypes: *specwcs*, *distortion*, *filteroffset*
WCS pipeline coordinate frames: grism_detector, detector, v2v3, world
Implements: reference files provided by NIRCam team

NIS_IMAGE, NIS_TACQ, NIS_TACONFIRM, NIS_FOCUS

reftypes: *distortion*, *filteroffset*
WCS pipeline coordinate frames: detector, v2v3, world
Implements: reference file provided by NIRISS team

NIS_WFSS

reftypes: *specwcs*, *distortion*, *filteroffset*
WCS pipeline coordinate frames: grism_detector, detector, v2v3, world
Implements: reference files provided by NIRISS team

NIS_SOSS

reftypes: *distortion*, *specwcs*
WCS pipeline coordinate frames: detector, v2v3, world
Implements: reference files provided by NIRISS team

NRS_FIXEDSLIT, NRS_MSASPEC, NRS_LAMP, NRS_BRIGHTOBJ

reftypes: *fpa*, *camera*, *disperser*, *collimator*, *msa*, *wavelengthrange*, *fore*, *ote*
WCS pipeline coordinate frames: detector, sca, bgwa, slit_frame, msa_frame, ote, v2v3, world
Implements: CDP 3 delivery

NRS_IFU

reftypes: *fpa*, *camera*, *disperser*, *collimator*, *msa*, *wavelengthrange*, *fore*, *ote*, *ifufore*, *ifuslicer*, *ifupost*
WCS pipeline coordinate frames: detector, sca, bgwa, slit_frame, msa_frame, ote, v2v3, world
Implements: CDP 3 delivery

NRS_IMAGING, NRS_MIMF, NRS_BOTA, NRS_CONFIRM, NRS_TACONFIRM, NRS_TASLIT, NRS_TACQ

reftypes: *fpa*, *camera*, *disperser*, *collimator*, *msa*, *wavelengthrange*, *fore*, *ote*
WCS pipeline coordinate frames: detector, sca, bgwa, slit_frame, msa_frame, ote, v2v3, world
Implements: CDP 3 delivery

jwst.assign_wcs Package

Functions

nrs_wcs_set_input(input_model, slit_name[, ...])	Returns a WCS object for a specific slit, slice or shutter.
nrs_ifu_wcs(input_model)	Return a list of WCSs for all NIRSPEC IFU slits.
get_spectral_order_wrange(input_model, ...)	Read the spectral order and wavelength range from the reference file.
niriss_soss_set_input(model, order_number)	Extract a WCS fr a specific spectral order.
update_fits_wcsinfo(datamodel[, ...])	Update `datamodel.meta.wcsinfo` based on a FITS WCS + SIP approximation of a GWCS object.

nrs_wcs_set_input

jwst.assign_wcs.**nrs_wcs_set_input**(*input_model*, *slit_name*, *wavelength_range=None*, *slit_y_low=None*, *slit_y_high=None*)

Returns a WCS object for a specific slit, slice or shutter.

Parameters

- **input_model** (JwstDataModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel – A WCS object for the all open slitlets in an observation.
- **slit_name** (*int* (https://docs.python.org/3/library/functions.html#int) *or* *str* (https://docs.python.org/3/library/stdtypes.html#str)) – Slit.name of an open slit.
- **wavelength_range** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – Wavelength range for the combination of filter and grating.

Returns

wcsobj – WCS object for this slit.

Return type

WCS (https://gwcs.readthedocs.io/en/stable/api/gwcs.wcs.WCS.html#gwcs.wcs.WCS)

nrs_ifu_wcs

jwst.assign_wcs.**nrs_ifu_wcs**(*input_model*)

Return a list of WCSs for all NIRSPEC IFU slits.

Parameters

input_model (*jwst.datamodels.JwstDataModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels – The data model. Must have been through the assign_wcs step.

get_spectral_order_wrange

jwst.assign_wcs.**get_spectral_order_wrange**(*input_model*, *wavelengthrange_file*)

Read the spectral order and wavelength range from the reference file.

Parameters

- **input_model** (JwstDataModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel – The input data model.
- **wavelengthrange_file** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Reference file of type "wavelengthrange".

niriss_soss_set_input

jwst.assign_wcs.**niriss_soss_set_input**(*model*, *order_number*)

Extract a WCS fr a specific spectral order.

Parameters

- **model** (ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.data – An instance of an ImageModel

- **order_number** (*int* (https://docs.python.org/3/library/functions.html#int)) – the spectral order

Return type
WCS - the WCS corresponding to the spectral order.

update_fits_wcsinfo

jwst.assign_wcs.**update_fits_wcsinfo**(*datamodel*, *max_pix_error=0.01*, *degree=None*, *npoints=32*, *crpix=None*, *projection='TAN'*, *imwcs=None*, ***kwargs*)

Update `datamodel.meta.wcsinfo` based on a FITS WCS + SIP approximation of a GWCS object. By default, this function will approximate the datamodel's GWCS object stored in `datamodel.meta.wcs` but it can also approximate a user-supplied GWCS object when provided via the `imwcs` parameter.

The default mode in using this attempts to achieve roughly 0.01 pixel accuracy over the entire image.

This function uses the `to_fits_sip()` (https://gwcs.readthedocs.io/en/stable/api/gwcs.wcs.WCS.html#gwcs.wcs.WCS.to_fits_sip to create FITS WCS representations of GWCS objects. Only most important `to_fits_sip()` (https://gwcs.readthedocs.io/en/stable/api/gwcs.wcs.WCS.html#gwcs.wcs.WCS.to_fits_sip) parameters are exposed here. Other arguments to `to_fits_sip()` (https://gwcs.readthedocs.io/en/stable/api/gwcs.wcs.WCS.html#gwcs.wcs.WCS.to_fits_sip) can be passed via `kwargs` - see "Other Parameters" section below. Please refer to the documentation of `to_fits_sip()` (https://gwcs.readthedocs.io/en/stable/api/gwcs.wcs.WCS.html#gwcs.wcs.WCS.to_fits_sip) for more details.

> **Warning:** This function modifies input data model's `datamodel.meta.wcsinfo` members.

Parameters

- **datamodel** (`ImageModel`) – The input data model for imaging or WFSS mode whose `meta.wcsinfo` field should be updated from GWCS. By default, `datamodel.meta.wcs` is used to compute FITS WCS + SIP approximation. When `imwcs` is not `None` (https://docs.python.org/3/library/constants.html#None) then computed FITS WCS will be an approximation of the WCS provided through the `imwcs` parameter.
- **max_pix_error** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Maximum allowed error over the domain of the pixel array. This error is the equivalent pixel error that corresponds to the maximum error in the output coordinate resulting from the fit based on a nominal plate scale.
- **degree** (*int* (https://docs.python.org/3/library/functions.html#int), *iterable*, *None*, *optional*) – Degree of the SIP polynomial. Default value `None` (https://docs.python.org/3/library/constants.html#None) indicates that all allowed degree values (`[1...6]`) will be considered and the lowest degree that meets accuracy requerements set by `max_pix_error` will be returned. Alternatively, `degree` can be an iterable containing allowed values for the SIP polynomial degree. This option is similar to default `None` (https://docs.python.org/3/library/constants.html#None) but it allows caller to restrict the range of allowed SIP degrees used for fitting. Finally, `degree` can be an integer indicating the exact SIP degree to be fit to the WCS transformation. In this case `max_pixel_error` is ignored.
- **npoints** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – The number of points in each dimension to sample the bounding box for use in the SIP fit. Minimum number of points is 3.

- **crpix** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of float* (https://docs.python.org/3/library/functions.html#float), *None, optional*) – Coordinates (1-based) of the reference point for the new FITS WCS. When not provided, i.e., when set to None (https://docs.python.org/3/library/constants.html#None) (default) the reference pixel already specified in `wcsinfo` will be re-used. If `wcsinfo` does not contain `crpix` information, then the reference pixel will be chosen near the center of the bounding box for axes corresponding to the celestial frame.
- **projection** (str, `Pix2SkyProjection`, optional) – Projection to be used for the created FITS WCS. It can be specified as a string of three characters specifying a FITS projection code from Table 13 in Representations of World Coordinates in FITS (https://doi.org/10.1051/0004-6361:20021326) (Paper I), Greisen, E. W., and Calabretta, M. R., A & A, 395, 1061-1075, 2002. Alternatively, it can be an instance of one of the astropy's Pix2Sky_* (https://docs.astropy.org/en/stable/modeling/reference_api.html#module-astropy.modeling.projections) projection models inherited from `Pix2SkyProjection`.
- **imwcs** (`gwcs.WCS`, None, optional) – Imaging GWCS object for WFSS mode whose FITS WCS approximation should be computed and stored in the `datamodel.meta.wcsinfo` field. When `imwcs` is None (https://docs.python.org/3/library/constants.html#None) then WCS from `datamodel.meta.wcs` will be used.

  Warning: Used with WFSS modes only. For other modes, supplying a different WCS from `datamodel.meta.wcs` will result in the GWCS and FITS WCS descriptions to diverge.

- **max_inv_pix_error** (*float* (https://docs.python.org/3/library/functions.html#float), *None, optional*) – Maximum allowed inverse error over the domain of the pixel array in pixel units. With the default value of None (https://docs.python.org/3/library/constants.html#None) no inverse is generated.
- **inv_degree** (*int* (https://docs.python.org/3/library/functions.html#int), *iterable, None, optional*) – Degree of the SIP polynomial. Default value None (https://docs.python.org/3/library/constants.html#None) indicates that all allowed degree values (`[1...6]`) will be considered and the lowest degree that meets accuracy requerements set by `max_pix_error` will be returned. Alternatively, `degree` can be an iterable containing allowed values for the SIP polynomial degree. This option is similar to default None (https://docs.python.org/3/library/constants.html#None) but it allows caller to restrict the range of allowed SIP degrees used for fitting. Finally, `degree` can be an integer indicating the exact SIP degree to be fit to the WCS transformation. In this case `max_inv_pixel_error` is ignored.
- **bounding_box** (*tuple* (https://docs.python.org/3/library/stdtypes.html#tuple), *None, optional*) – A pair of tuples, each consisting of two numbers Represents the range of pixel values in both dimensions ((xmin, xmax), (ymin, ymax))
- **verbose** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Print progress of fits.

Return type

FITS header with all SIP WCS keywords

Raises

`ValueError` (https://docs.python.org/3/library/exceptions.html#ValueError) – If the WCS is not at least 2D, an exception will be raised. If the specified accuracy (both forward and inverse, both rms and maximum) is not achieved an exception will be raised.

Notes

Use of this requires a judicious choice of required accuracies. Attempts to use higher degrees (~7 or higher) will typically fail due to floating point problems that arise with high powers.

For more details, see `to_fits_sip()` (https://gwcs.readthedocs.io/en/stable/api/gwcs.wcs.WCS.html#gwcs.wcs.WCS.to_fits_sip)

Classes

AssignWcsStep([name, parent, config_file, ...])	AssignWcsStep: Create a gWCS object and store it in `Model.meta`.

AssignWcsStep

class jwst.assign_wcs.**AssignWcsStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

AssignWcsStep: Create a gWCS object and store it in `Model.meta`.

Reference file types:

camera Camera model (NIRSPEC) collimator Collimator Model (NIRSPEC) disperser Disperser model (NIRSPEC) distortion Spatial distortion model (FGS, MIRI, NIRCAM, NIRISS) filteroffset Filter offsets (MIRI Imager) fore Transform through the FORE optics (NIRSPEC) fpa Transform in the FPA plane (NIRSPEC) ifufore Transforms from the MSA plane to the plane of the IFU slicer (NIRSPEC) ifupost Transforms from the slicer plane to the MSA plane (NIRSPEC) ifuslicer Metrology of the IFU slicer (NIRSPEC) msa Metrology of the MSA plane (NIRSPEC) ote Transform through the Optical Telescope Element (NIRSPEC) specwcs Wavelength calibration models (MIRI, NIRCAM, NIRISS) regions Stores location of the regions on the detector (MIRI) wavelengthrange Typical wavelength ranges (MIRI, NIRCAM, NIRISS, NIRSPEC)

Parameters

input (`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamo
`IFUImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamo
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.Cube
– Input exposure.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input, *args, **kwargs)	This is where real work happens.

Attributes Documentation

class_alias = 'assign_wcs'

reference_file_types = ['distortion', 'filteroffset', 'specwcs', 'regions', 'wavelengthrange', 'camera', 'collimator', 'disperser', 'fore', 'fpa', 'msa', 'ote', 'ifupost', 'ifufore', 'ifuslicer']

spec

```
sip_approx = boolean(default=True)  # enables SIP approximation for imaging␣
↪modes.
sip_max_pix_error = float(default=0.1)  # max err for SIP fit, forward.
sip_degree = integer(max=6, default=None)  # degree for forward SIP fit, None␣
↪to use best fit.
sip_max_inv_pix_error = float(default=0.1)  # max err for SIP fit, inverse.
sip_inv_degree = integer(max=6, default=None)  # degree for inverse SIP fit,␣
↪None to use best fit.
sip_npoints = integer(default=12)  #  number of points for SIP
slit_y_low = float(default=-.55)  # The lower edge of a slit.
slit_y_high = float(default=.55)  # The upper edge of a slit.
```

Methods Documentation

process(*input*, **args*, ***kwargs*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.7 Associations

Association Overview

What are Associations?

Associations are basically just lists of things, mostly exposures, that are somehow related. With respect to JWST and the Data Management System (DMS), associations have the following characteristics:

- Relationships between multiple exposures are captured in an association.
- An association is a means of identifying a set of exposures that belong together and may be dependent upon one another.
- The association concept permits exposures to be calibrated, archived, retrieved, and reprocessed as a set rather than as individual objects.
- For each association, DMS will generate the most combined and least combined data products.

Associations and JWST

The basic chunk in which science data arrives from the observatory is termed an *exposure*. An exposure contains the data from a single set of integrations per detector per instrument. In general, it takes many exposures to make up a single observation, and a whole program is made up of a large number of observations.

On first arrival, an exposure is termed to be at *Level1b*: The only transformation that has occurred is the extraction of the science data from the observatory telemetry into a FITS file. At this point, the science exposures enter the calibration pipeline.

The pipeline consists of three stages: Stage 1, Stage 2, and Stage 3 processing. Stage 2 processing is the calibration necessary to remove instrumental effects from the data. The resulting files contain flux and spatially calibrated data, called *Stage 2b* data. The information is still in individual exposures.

Note: Older documentation and code may refer to the stages as **levels**. They are synonymous.

To be truly useful, the exposures need to be combined and, in the case of multi-object spectrometry, separated, into data that is source-oriented. This type of calibration is called *Stage 3* processing. Due to the nature of the individual instruments, observing modes, and the interruptibility of the observatory itself, how to group the right exposures together is not straight-forward.

Enter the *Association Generator*. Given a set of exposures, called the *Association Pool*, and a set of rules found in an *Association Registry*, the generator groups the exposures into individual *associations*. These associations are then

used as input to the Stage 3 calibration steps to perform the transformation from exposure-based data to source-based, high(er) signal-to-noise data.

In short, Stage 2 and Stage 3 associations are created running the *asn_generate* task on an `AssociationPool` using the default *Level2* and *Level 3* association rules to produce Stage 2 and Stage 3 associations. When retrieving the data from the archive, users will find the list of associated data in JSON files that are submitted together with the requested Stage 2 or Stage 3 data.

Association Pools

The information about what data will be associated is constructed with the information derived from the Astronomer Proposal Tool and the rules on how data should be associated that are defined by the instrument teams. All the information from a single proposal is captured in a single file known as the *Association Pool*.

Usage

Users should not need to run the generator. Instead, it is expected that one edits an already existing association that accompanies the user's JWST data. Care should be taken if editing an association file. Keep in mind all input files listed in the association file are in the same directory as the association file and no path information can be put in `expname`, only the file name. Or, if need be, an association can be created based on the existing *Stage 2* or *Stage 3* examples. If, however, the user *does* need to run the generator, *Association Generator* documentation will be helpful.

Once an association is in-hand, one can pass it as input to a pipeline routine. For example:

```
% strun calwebb_image3  jw12345-o001_20210311t170002_image3_001_asn.json
```

Programmatically, to read in an Association, one uses the *load_asn()* function:

```
from jwst.associations import load_asn

with open('jw12345-o001_20210311t170002_image3_001_asn.json') as fp:
    asn = load_asn(fp)
```

What exactly is returned depends on what the association is. However, for all Stage 2 and Stage 3 associations, a Python `dict` is returned, whose structure matches that of the JSON or YAML file. Continuing from the above example, the following shows how to access the first exposure file name of a Stage 3 associations:

```
exposure = asn['products'][0]['members'][0]['expname']
```

Since most JWST data are some form of a *JWST Data Model* an association can be opened with datamodels.open (https://stdatamodels.readthedocs.io/en/latest/jwst/datamodels/models.html#datamodels-open) which returns a `ModelContainer`. All members of the association that can be represented as a `DataModel`, will be available in the `ModelContainer` as their respective DataModels.

```
from stdatamodels.jwst.datamodels import open as dm_open
container_model = dm_open('jw12345-o001_20210311t170002_image3_001_asn.json')
```

Utilities

There are a number of utilities to create user-specific associations that are documented under *Association Commands*.

JWST Associations

JWST Conventions

Association Naming

When produced through the ground processing, all association files are named according to the following scheme:

```
jwPPPPP-TNNNN_YYYYMMDDtHHMMSS_ATYPE_MMMMM_asn.json
```

where:

- `jw`: All JWST-related products begin with `jw`
- `PPPPP`: 5 digit proposal number
- `TNNNN`: Candidate Identifier. Can be one of the following:
  - `oNNN`: Observation candidate specified by the letter `o` followed by a 3 digit number.
  - `c1NNN`: Association candidate, specified by the letter 'c', followed by a number starting at 1001.
  - `a3NNN`: Discovered whole program associations, specified by the letter 'a', followed by a number starting at 3001
  - `rNNNN`: Reserved for future use. If you see this in practice, file an issue to have this document updated.
- `YYYYMMDDtHHMMSS`: This is generically referred to as the `version_id`. DMS specifies this as a timestamp. Note: When used outside the workflow, this field is user-specifiable.
- `ATYPE`: The type of association. See *Association Types*
- `MMMMM`: A counter for each type of association created.

Association Types

Each association is intended to make a specific science product. The type of science product is indicated by the `ATYPE` field in the association file name (see *Association Naming*), and in the `asn_type` meta keyword of the association itself (see *Association Meta Keywords*).

The pipeline uses this type as the key to indicate which Level 2 or Level 3 pipeline module to use to process this association.

The current association types are:

- `ami3`: Intended for *calwebb_ami3* processing
- `coron3`: Intended for *calwebb_coron3* processing
- `image2`: Intended for *calwebb_image2* processing
- `image3`: Intended for *calwebb_image3* processing
- `nrslamp-spec2`: Intended for *calwebb_spec2* processing
- `spec2`: Intended for *calwebb_spec2* processing

- `spec3`: Intended for *calwebb_spec3* processing
- `tso3`: Intended for *calwebb_tso3* processing
- `tso-image2`: Intended for *calwebb_image2* processing
- `tso-spec2`: Intended for *calwebb_spec2* processing
- `wfs-image2`: Intended for *calwebb_image2* processing
- `wfs-image3`: Intended for *calwebb_wfs-image3* processing

Field Guide to File Names

The high-level distinctions between stage 2, stage 3, exposure-centric, and target-centric files can be determined by the following file patterns. These patterns are not intended to fully define all the specific types of files there are. However, these are the main classifications, from which the documentation for the individual calibrations steps and pipelines will describe any further details.

The most general regex matches all files that have been produced by Stage 3 processing:

```
.+[aocr][0-9]{3:4}.+
```

The following regexes differentiate between exposure-centric and target-centric files.

- Files containing exposure-centric data

  The following regex matches files names produced by either Stage 2 or 3 calibration and containing exposure-centric data:

  ```
  jw[0-9]{11}_[0-9]{5}_[0-9]{5}_.+\.fits
  ```

- Files containing target-centric data

  The following regex matches file names produced by Stage 3 calibration and containing target-centric data:

  ```
  jw[0-9]{5}-[aocr][0-9]{3:4}_.+
  ```

Science Data Processing Workflow

General Workflow for Generating Association Products

See *Associations and JWST* for an overview of how JWST uses associations. This document describes how associations are used by the ground processing system to execute the stage 2 and stage 3 pipelines.

Up to the initial calibration step *calwebb_detector1*, the science exposures are treated individually. However, starting at the stage 2 calibration step, exposures may need other exposures in order to be further processed. Instead of creating a single monolithic pipeline, the workflow uses the associations to determine what pipeline should be executed and when to execute them. In the figure below, this wait-then-execute process is represented by the `workflow trigger`. The workflow reads the contents of an association file to determine what exposures, and possibly other files, are needed to continue processing. The workflow then waits until all exposures exist. At that point, the related calibration pipeline is executed with the association as input.

With this finer granularity, the workflow can run more processes parallel, and allows the operators deeper visibility into the progression of the data through the system.

The figure represents the following workflow:

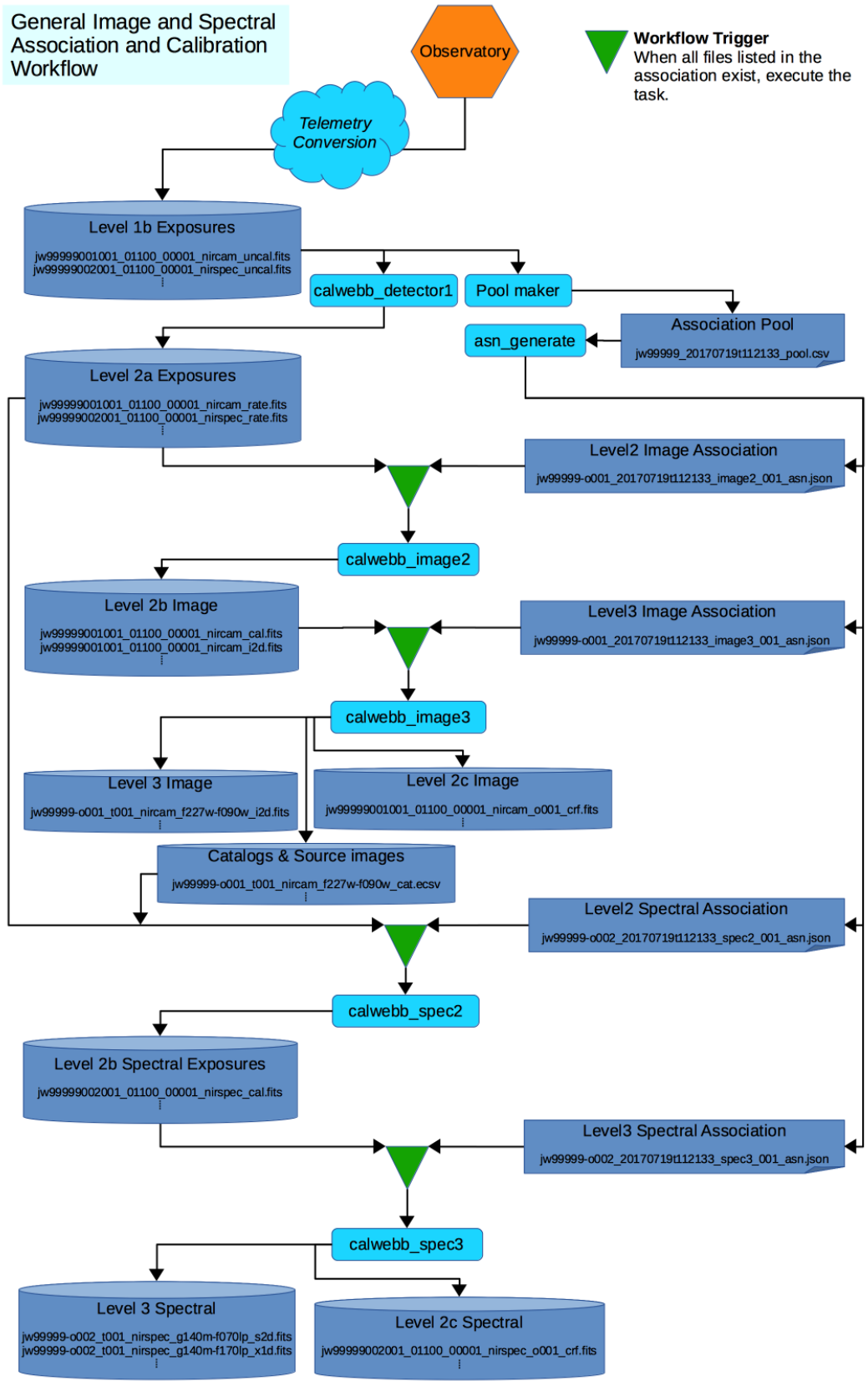


Fig. 1: General workflow through stage 2 and stage 3 processing

- Data comes down from the observatory and is converted to the raw FITS files.
- *calwebb_detector1* is run on each file to convert the data to the countrate format.
- In parallel with *calwebb_detector1*, the Pool Maker collects the list of downloaded exposures and places them in the Association Pool.
- When enough exposures have been download to complete an Association Candidate, such as an Observation Candidate, the Pool Maker calls the Association Generator, *asn_generate*, to create the set of associations based on that Candidate.
- For each association generated, the workflow creates a file watch list from the association, then waits until all exposures needed by that association come into existence.
- When all exposures for an association exist, the workflow then executes the corresponding pipeline, passing the association as input.

Wide Field Slitless Spectroscopy

In most cases, the data will flow from stage 2 to stage 3, completing calibration. However, more complicated situations can be handled by the same wait-then-execute process. One particular case is for the Wide Field Slitless Spectrometry (WFSS) modes. The specific flow is show in the figure below:

For WFSS data, at least two observations are made, one consisting of a direct image of the field-of-view (FOV), and a second where the FOV is dispersed using a grism. The direct image is first processed through stage 3. At the stage 3 stage, a source catalog of objects found in the image, and a segmentation map, used to record the minimum bounding box sizes for each object, are generated. The source catalog is then used as input to the stage 2 processing of the spectral data. This extra link between the two major stages is represented by the `Segment & Catalog` file set, shown in red in the diagram. The stage 2 association `grism_spec2_asn` not only lists the needed countrate exposures, but also the catalog file produced by the stage 3 image processing. Hence, the workflow knows to wait for this file before continuing the spectral processing.

Stage 2 Associations: Technical Specifications

Logical Structure

All stage 2 associations have the following structure. The structure is defined and enforced by the stage 2 schema.

- *Informational Meta Keywords*
- List of *products*, each consisting of
  - Output product name
  - List of *exposure members*, each consisting of
    - Filename of the exposure that is a member of this association
    - Type of exposure
    - If present, information about errors from the observatory log

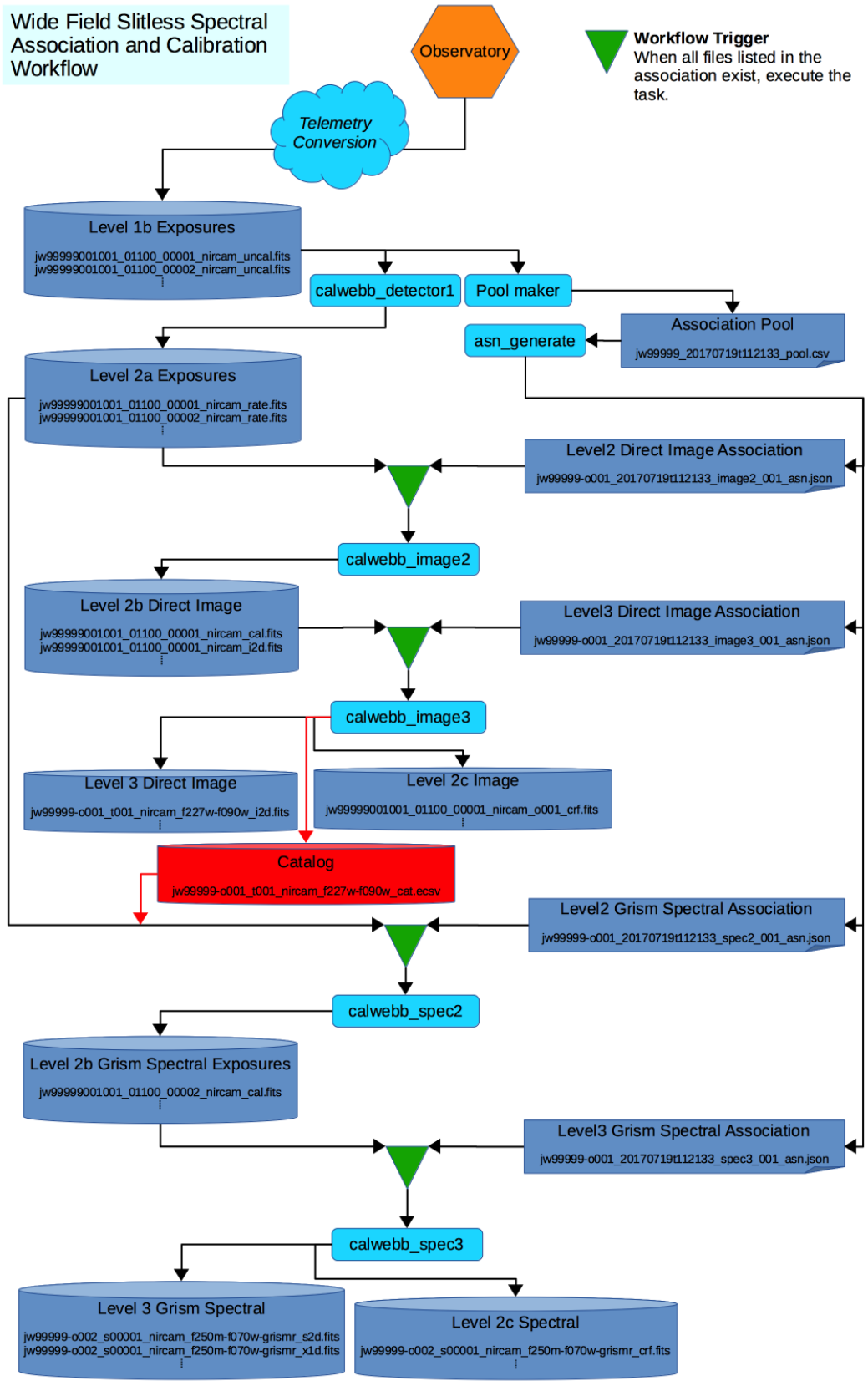


Fig. 2: WFSS workflow

Example Association

The following example will be used to explain the contents of an association:


```
{
    "asn_type": "image2",
    "asn_rule": "candidate_Asn_Lv2Image",
    "version_id": "20210610t121508",
    "code_version": "1.2.3",
    "degraded_status": "No known degraded exposures in association.",
    "program": "00623",
    "constraints": "DMSAttrConstraint({'name': 'program', 'sources': ['program'], 'value
↪': '623'})\nDMSAttrConstraint({'name': 'is_tso', 'sources': ['tsovisit'], 'value':␣
↪None})\nDMSAttrConstraint({'name': 'instrument', 'sources': ['instrume'], 'value':
↪'miri'})\nDMSAttrConstraint({'name': 'detector', 'sources': ['detector'], 'value':
↪'mirimage'})\nDMSAttrConstraint({'name': 'opt_elem', 'sources': ['filter'], 'value':
↪'f1130w'})\nDMSAttrConstraint({'name': 'opt_elem2', 'sources': ['pupil', 'grating'],
↪'value': None})\nDMSAttrConstraint({'name': 'opt_elem3', 'sources': ['fxd_slit'],
↪'value': None})\nDMSAttrConstraint({'name': 'subarray', 'sources': ['subarray'], 'value
↪': 'brightsky'})\nDMSAttrConstraint({'name': 'channel', 'sources': ['channel'], 'value
↪': None})\nDMSAttrConstraint({'name': 'slit', 'sources': ['fxd_slit'], 'value': None})\
↪nConstraint_Image_Science({'name': 'exp_type', 'sources': ['exp_type'], 'value': 'mir_
↪image'})\nConstraint_Single_Science({'name': 'single_science', 'value': False})\
↪nDMSAttrConstraint({'name': 'asn_candidate', 'sources': ['asn_candidate'], 'value': \"\
↪\\\('o037',\\\\ 'observation'\\\\)\"})",
    "asn_id": "o037",
    "asn_pool": "jw00623_20210610t121508_pool",
    "target": "9",
    "products": [
        {
            "name": "jw00623037001_02101_00001_mirimage",
            "members": [
                {
                    "expname": "jw00623037001_02101_00001_mirimage_rate.fits",
                    "exptype": "science",
                    "exposerr": "null"
                }
            ]
        }
    ]
}
```


Association Meta Keywords

The following are the informational, meta keywords of an association.

asn_id ***required***
: The association id. The id is what appears in the *Association Naming*

asn_pool ***required***
: Association pool from which this association was created.

asn_rule ***optional***
: Name of the association rule which created this association.

asn_type *optional*
The type of association represented. See *Association Types*

code_version *optional*
The version of the generator that created this association. Typically this is the version of the jwst python package.

constraints *optional*
List of constraints used by the association generator to create this association. Format and contents are determined by the defining rule.

degraded_status *optional*
If any of the included members have an actual issue, as reported by the `exposerr` keyword, `degraded_status` will have the value `One or more members have an error associated with them.` If no errors are indicated, the value will be `No known degraded exposures in association.`

program *optional*
Program number for which this association was created.

target *optional*
Target ID for which this association refers to. JWST currently uses the TARGETID header keyword in the stage 2 exposure files, but there is no formal restrictions on value.

version_id *optional*
Version identifier. DMS uses a time stamp with the format `yyyymmddthhmmss` Can be None or NULL

`products` Keyword

A list of products that would be produced by this association. For stage 2, each product is an exposure. Each product should have one `science` member, the exposure on which the stage 2 processing will occur.

Association products have two components:

name *optional*
The string template to be used by stage 2 processing tasks to create the output file names. The product name, in general, is a prefix on which the individual pipeline and step modules will append whatever suffix information is needed.

If not specified, the stage 2 processing modules will create a name based off the name of the `science` member.

members *required*
This is a list of the exposures to be used by the stage 2 processing tasks. This keyword is explained in detail in the next section.

`members` Keyword

`members` is a list of dictionaries, one for each member exposure in the association. Each member has the following keywords.

expname *required*
The exposure file name. This must be a filename only, with no path. This file must be in the same directory as the association file, so path is not needed. If path data is included, an exception is raised when loading the association file.

exptype *required*
Type of information represented by the exposure. Possible values are as follows. *Note that this is not the same as the exposure ``EXP_TYPE`` header keyword.*

- `science`: Primary science exposure. For each product, only one exposure can be `science`.

- `background`: Background exposure to subtract.
- `imprint`: Imprint exposure to subtract.
- `sourcecat`: The catalog of sources to extract spectra for. Usually produced by *calwebb_image3* for wide-field slitless spectroscopy.
- `segmap`: The 2D segmentation map used to produce the source catalog. Usually produced by *calwebb_image3* for wide-field slitless spectroscopy.
- `direct_image`: The direct image used to produce the source catalog.

Editing the member list

As discussed previously, a member is made up of a number of keywords, formatted as follows:

```
{
    "expname": "jw_00003_cal.fits",
    "exptype": "science",
},
```

To remove a member, simply delete its corresponding set.

To add a member, one need only specify the two required keywords:

```
{
    "expname": "jw_00003_cal.fits",
    "exptype": "science"
},
```

Stage 3 Associations: Technical Specifications

Logical Structure

Independent of the actual format, all stage 3 associations have the following structure. Again, the structure is defined and enforced by the stage 3 schema

- *Informational Meta Keywords*
- List of *products*, each consisting of
  - Output product name
  - List of *exposure members*, each consisting of
    - Filename of the exposure that is a member of this association
    - Type of exposure
    - If present, information about errors from the observatory log
    - If present, the Association Candidates this exposure belongs to

Example Association

The following example will be used to explain the contents of an association:


```
{
    "degraded_status": "No known degraded exposures in association.",
    "version_id": "20160826t131159",
    "asn_type": "image3",
    "asn_id": "c3001",
    "constraints": "Constraints:\n    opt_elem2: CLEAR\n    detector: (?!NULL).+\n
↪target_name: 1\n    exp_type: NRC_IMAGE\n    wfsvisit: NULL\n    instrument: NIRCAM\n
↪  opt_elem: F090W\n    program: 99009",
    "asn_pool": "mega_pool",
    "asn_rule": "Asn_Image",
    "target": "1",
    "program": "99009",
    "products": [
        {
            "name": "jw99009-a3001_t001_nircam_f090w",
            "members": [
                {
                    "exposerr": null,
                    "expname": "jw_00001_cal.fits",
                    "asn_candidate": "[('o001', 'observation')]",
                    "exptype": "science"
                },
                {
                    "exposerr": null,
                    "expname": "jw_00002_cal.fits",
                    "asn_candidate": "[('o001', 'observation')]",
                    "exptype": "science"
                }
            ]
        }
    ]
}
```


Association Meta Keywords

The following are the informational, meta keywords of an association.

asn_id ***required***
: The association id. The id is what appears in the *Association Naming*

asn_pool ***required***
: Association pool from which this association was created.

asn_rule ***optional***
: Name of the association rule which created this association.

asn_type ***optional***
: The type of association represented. See *Association Types*

code_version ***optional***
: The version of the generator that created this association. Typically this is the version of the jwst python package.

constraints *optional*
List of constraints used by the association generator to create this association. Format and contents are determined by the defining rule.

degraded_status *optional*
If any of the included members have an actual issue, as reported by the `exposerr` keyword, `degraded_status` will have the value `One or more members have an error associated with them.` If no errors are indicated, the value will be `No known degraded exposures in association.`

program *optional*
Program number for which this association was created.

target *optional*
Target ID to which this association refers. JWST currently uses the TARGETID header keyword in the stage 2 exposure files, but there are no formal restrictions on value.

version_id *optional*
Version identifier. DMS uses a time stamp with the format `yyyymmddthhmmss` Can be None or NULL

`products` Keyword

Association products have two components:

name *optional*
The string template to be used by stage 3 processing tasks to create the output file names. The product name, in general, is a prefix on which the individual pipeline and step modules will append whatever suffix information is needed.

If not specified, the stage 3 processing modules will create a name root.

members *required*
This is a list of the exposures to be used by the stage 3 processing tasks. This keyword is explained in detail in the next section.

`members` Keyword

`members` is a list of dictionaries, one for each member exposure in the association. Each member has the following keywords.

expname *required*
The exposure file name

exptype *required*
Type of information represented by the exposure. Possible values are

- `science`: The primary science exposures. There is usually more than one since stage 3 calibration involves combining multiple science exposures. However, at least one exposure in an association needs to be `science`.
- `background`: Exposures used for background subtraction.
- `psf`: Exposures that should be considered PSF references for coronagraphic and AMI calibration.

exposerr *optional*
If there was some issue the occurred on the observatory that may have affected this exposure, that condition is listed here. Otherwise the value is `null`

asn_candidate *optional*
Contains the list of association candidates this exposure belongs to.

Editing the member list

As discussed previously, a member is made up of a number of keywords, formatted as follows:

```
{
    "expname": "jw_00003_cal.fits",
    "exptype": "science",
    "exposerr": null,
    "asn_candidate": "[('o001', 'observation')]"
},
```

To remove a member, simply delete its corresponding set.

To add a member, one need only specify the two required keywords:

```
{
    "expname": "jw_00003_cal.fits",
    "exptype": "science"
},
```

Stage 2 Associations: Rules

The following table describes exactly which exposures will go through any type of stage 2 processing, such as Spec2Pipeline or Image2Pipeline.

Table 1: Exposure Modes for Stage 2 Processing

EXP_TYPE	Member Exposure Type	Specials	Association Type
FGS_ACQ1	tracking	N/A	N/A
FGS_ACQ2	tracking	N/A	N/A
FGS_DARK	dark	N/A	N/A
FGS_FINEGUIDE	tracking	N/A	N/A
FGS_FOCUS	science	N/A	image2
FGS_ID-IMAGE	tracking	N/A	N/A
FGS_ID-STACK	tracking	N/A	N/A
FGS_IMAGE	science	N/A	image2
FGS_INTFLAT	flat	N/A	N/A
FGS_SKYFLAT	flat	N/A	N/A
FGS_TRACK	tracking	N/A	N/A
MIR_4QPM	psf	PSF	image2
MIR_4QPM	science	N/A	image2
MIR_CORONCAL	science	N/A	image2
MIR_DARKIMG	dark	N/A	N/A
MIR_DARKMRS	dark	N/A	N/A
MIR_FLATIMAGE	flat	N/A	N/A
MIR_FLATIMAGE-EXT	flat	N/A	N/A
MIR_FLATMRS	flat	N/A	N/A
MIR_FLATMRS-EXT	flat	N/A	N/A
MIR_IMAGE	science	N/A	image2
MIR_LRS-FIXEDSLIT	background	BACKGROUND	spec2
MIR_LRS-FIXEDSLIT	science	N/A	spec2

continues on next page

Table 1 – continued from previous page

EXP_TYPE	Member Exposure Type	Specials	Association Type
MIR_LRS-SLITLESS	background	BACKGROUND	spec2
MIR_LRS-SLITLESS	science	N/A	spec2
MIR_LYOT	psf	PSF	image2
MIR_LYOT	science	N/A	image2
MIR_MRS	background	BACKGROUND	spec2
MIR_MRS	science	N/A	spec2
MIR_TACQ	target_acquisition	N/A	image2
NIS_AMI	psf	PSF	image2
NIS_AMI	science	N/A	image2
NIS_DARK	science	N/A	N/A
NIS_EXTCAL	science	N/A	N/A
NIS_FOCUS	science	N/A	image2
NIS_IMAGE	science	N/A	images
NIS_LAMP	science	N/A	N/A
NIS_SOSS	science	N/A	spec2
NIS_TACONFIRM	target_acquisition	N/A	image2
NIS_TACQ	target_acquisition	N/A	image2
NIS_WFSS	science	N/A	spec2
NRC_CORON	psf	PSF	image2
NRC_CORON	science	N/A	image2
NRC_DARK	dark	N/A	N/A
NRC_FLAT	flat	N/A	N/A
NRC_FOCUS	science	N/A/	image2
NRC_GRISM	science	N/A	N/A
NRC_IMAGE	science	N/A	image2
NRC_LED	science	N/A	N/A
NRC_TACONFIRM	target_acquisition	N/A	image2
NRC_TACQ	target_acquisition	N/A	image2
NRC_TSGRISM	science	N/A	tso-spec2
NRC_TSIMAGE	science	N/A	tso-image2
NRC_WFSS	science	N/A	spec2
NRS_AUTOFLAT	nrs_autoflat	N/A	image2
NRS_AUTOWAVE	nrs_autowave	N/A	image2
NRS_BRIGHTOBJ	science	N/A	spec2
NRS_CONFIRM	science	N/A	image2
NRS_DARK	dark	N/A	N/A
NRS_FIXEDSLIT	background	BACKGROUND	spec2
NRS_FIXEDSLIT	science	N/A	spec2
NRS_FOCUS	science	N/A	image2
NRS_IFU	background	BACKGROUND	spec2
NRS_IFU	imprint	IMPRINT	spec2
NRS_IFU	science	N/A	spec2
NRS_IMAGE	science	N/A	image2
NRS_LAMP[1]	science	N/A	nrslamp-spec2
NRS_MIMF	science	N/A	wfs-image2
NRS_MSASPEC	imprint	IMPRINT	spec2
NRS_MSASPEC	science	N/A	spec2

continues on next page

Table 1 – continued from previous page

EXP_TYPE	Member Exposure Type	Specials	Association Type
NRS_MSATA	target_acquisition	N/A	image2
NRS_TACONFIRM	target_acquisition	N/A	image2
NRS_VERIFY	science	N/A	image2
NRS_WATA	target_acquisition	N/A	image2

Footnotes

Notes

Column definitions

- EXP_TYPE : The exposure type.
- Member Exposure Type: How the association generator will classify the exposure.
- Specials : The association rule modifications to handle the exposure.
- Association Type : *Association type* created.

More about Specials: Many exposures that are not directly science, such as backgrounds, are primarily used as auxiliary members for other science products. However, they are also often calibrated as if they were science products themselves. In these situations, a special association rule is created to produce the necessary associations.

History

The original content of this page is from github issue #1188 (https://github.com/spacetelescope/jwst/issues/1188).

Stage3 Associations: Rules

Data Grouping

JWST exposures are identified and grouped in a specific order, as follows:

- program

  The entirety of a science observing proposal is contained within a `program`. All observations, regardless of instruments, pertaining to a proposal are identified by the program id.

- observation

  A set of visits, any corresponding auxiliary exposures, such as wavelength calibration, using a specific instrument. An observation does not necessarily contain all the exposures required for a specific observation mode. Also, exposures within an observation can be taken with different optical configurations of the same instrument

- visit

  A set of exposures which sharing the same source, or target, whether that would be external to the observatory or internal to the instrument. The can be many visits for the same target, and visits to different targets can be interspersed among themselves.

[1] Association creation is heavily dependent upon other parameters such as `LAMP`, `OPMODE`, and `GRATING`.

- group

  A set of exposures that share the same observatory configuration. This is basically a synchronization point between observatory moves and parallel instrument observations.

- sequence

  TBD

- activity

  A set of exposures that are to be taken atomically. All exposures within an activity are associated with each other and have been taken consecutively.

- exposure

  The basic unit of science data. Starting at stage 1, an exposure contains a single integrations of a single detector from a single instrument for a single *snap*. Note that a single integration actually is a number of readouts of the detector during the integration.

Rules

All rules have as their base class `DMS_Level3_Base` This class defines the association structure, enforces the DMS naming conventions, and defines the basic validity checks on the Level3 associations.

Along with the base class, a number of mixin classes are defined. These mixins define some basic constraints that are found in a number of rules. An example is the `AsnMixin_Base`, which provides the constraints that ensure that the program identification and instrument are the same in each association.

The rules themselves are subclasses of `AsnMixin_Base` and whatever other mixin classes are necessary to build the rule. Conforming to the *Class Naming* scheme, all the final Level3 association rules begin with `Asn_`. An example is the `Asn_Image` rule.

The following figure shows the above relationships. Note that this diagram is not meant to be a complete listing.

Design

Association Design

As introduced in the *Association Overview*, the figure above shows all the major players used in generating associations. Since this section will be describing the code design, the figure below is the overview but using the class names involved.

Generator

Algorithm

The generator conceptual workflow is show below:

This workflow is encapsulated in the *`generate()`* function. Each member is first checked to see if it belongs to an already existing association. If so, it is added to each association it matches with. Next, the set of association rules are check to see if a new association, or associations, are created by the member. However, only associations that have not already been created are checked for. This is to prevent cyclical creation of associations.

As discussed in *Associations and Rules*, associations are Python classes, often referred to as `association rules`, and their instantiations, referred to as `associations`. An association is created by calling the *`Association.create`* class method for each association rule. If the member matches the rule, an association is returned. Each

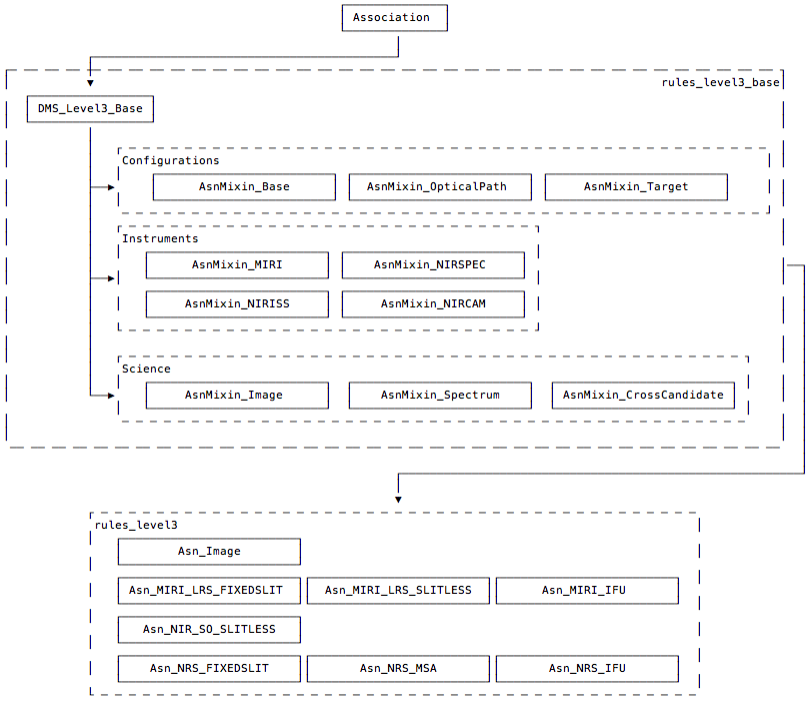


Fig. 3: Level3 Rule Class Inheritance

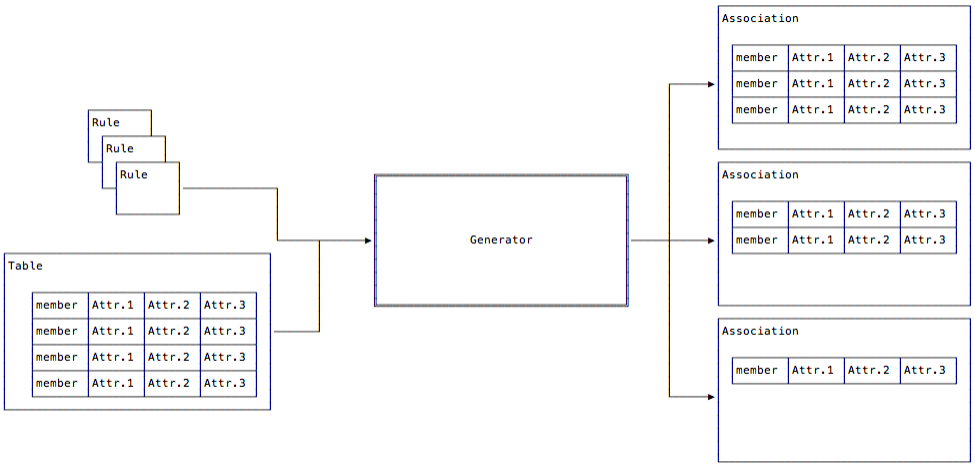


Fig. 4: Association Generator Overview

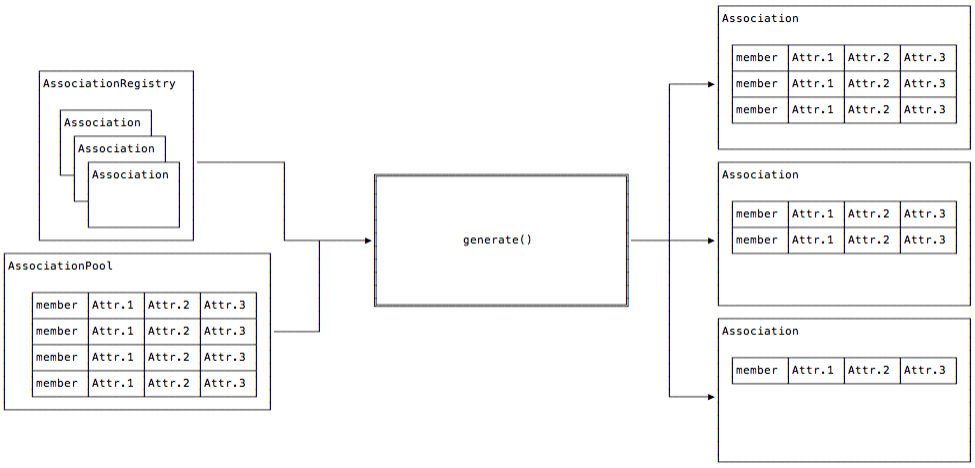


Fig. 5: Association Class Relationship overview

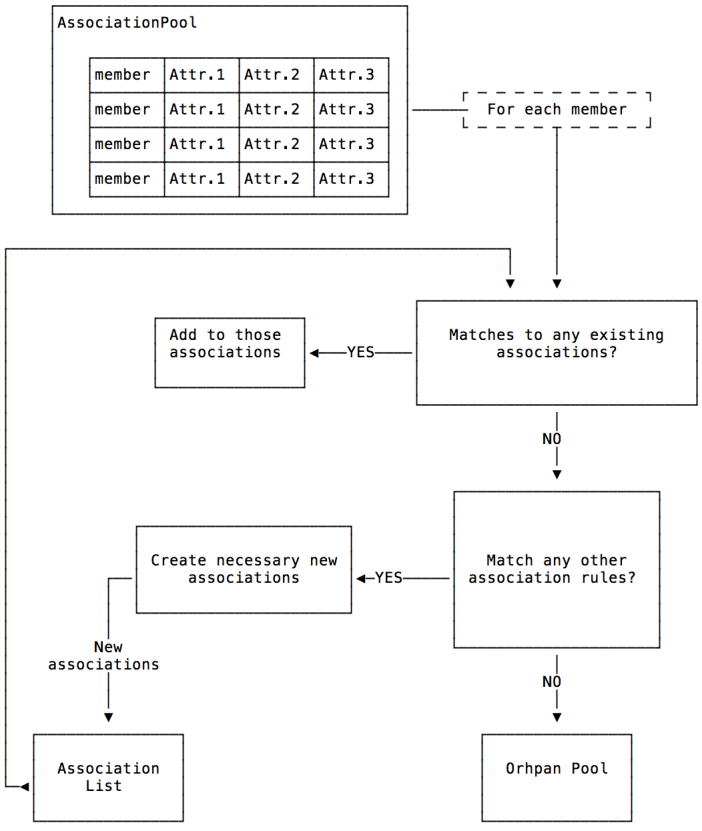


Fig. 6: Generator Conceptual Workflow

defined rule tried. This process of checking whether a member would create any associations is encapsulated in the *AssociationRegistry.match* method

Conversely, to see if a member belongs to an already existing association, an attempt is made to add the member using the *Association.add* method. If the addition succeeds, the member has been added to the association instance. The generator uses `match_member` function to loop through its list of existing associations.

Output

Before exiting, *generate()* checks the *Association.is_valid* property of each association to ensure that an association has all the members it is required to have. For example, if a JWST coronagraphic observation was performed, but the related PSF observation failed, the coronagraphic association would be marked invalid.

Once validation is complete, *generate()* returns a 2-tuple. The first item is a list of the associations created. The second item is another *AssociationPool* containing all the members that did not get added to any association.

Member Attributes that are Lists

As mentioned in *Association Pool*, most member attributes are simply treated as strings. The exception is when an attribute value looks like a list:

```
[element, ...]
```

When this is the case, a *mini pool* is created. This pool consists of duplicates of the original member. However, for each copy of the member, the attribute that was the list is now populated with consecutive members of that list. This mini pool and the rule or association in which this was found, is passed back up to the *generate()* function to be reconsidered for membership. Each value of the list is considered separately because association membership may depend on what those individual values are. The figure below demonstrates the member replication.

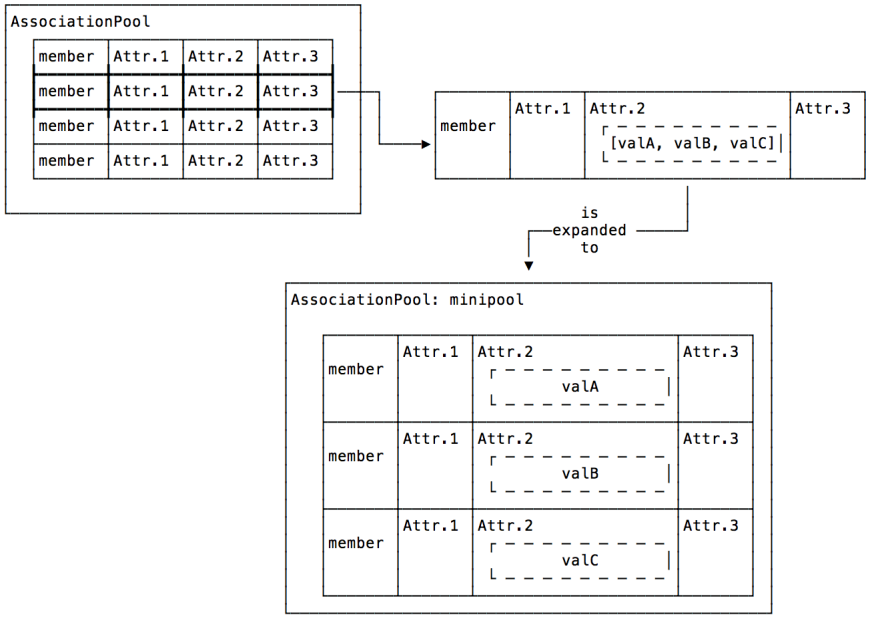


Fig. 7: Member list expansion
Attr.2 is a list of three values which expands into three members in the mini pool.

For JWST, this is used to filter through the various types of association candidates. Since an exposure can belong to more than one association candidate, the exposure can belong to different associations depending on the candidates.

Association Candidates

TBD

Associations and Rules

Terminology

As *has been described*, an `Association` is a Python dict or list that is a list of things that belong together and are created by association rules. However, as will be described, the association rules are Python classes which inherit from the `Association` class.

Associations created from these rule classes, refered to as just `rules`, have the type of the class they are created from and have all the methods and attributes of those classes. Such instances are used to populate the created associations with new members and check the validity of said associations.

However, once an association has been saved, or serialized, through the *`Association.dump`* method, then reload through the corresponding *`Association.load`* method, the restored association is only the basic list or dict. The whole instance of the originating association is not serialized with the basic membership information.

This relationship is shown in the following figure:

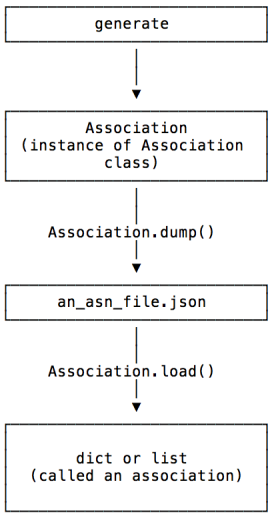


Fig. 8: Rule vs. Association Relationship

Note About Loading

Association.load will only validate the incoming data against whatever schema or other validation checks the particular subclass calls for. The generally preferred method for loading an association is through the *jwst.associations.load_asn()* function.

Rules

Association rules are Python classes which must inherit from the *Association* base class. What the rules do and what they create are completely up to the rules themselves. Except for a few *core methods*, the only other requirement is that any instance of an association rule must behave as the association it creates. If the association is a dict, the rule instance must behave as the dict. If the association is a list, the rule instance must behave as a list. Otherwise, any other methods and attributes the rules need for association creation may be added.

Rule Sets

In general, because a set of rules will share much the same functionality, for example how to save the association and how to decide membership, it is suggested that an intermediate set of classes be created from which the rule classes inherit. The set of rule classes which share the same base parent classes are referred to as a *rule set*. The JWST *Level 2* and *Level 3* are examples of such rule sets. The below figure demonstrates the relationships between the base `Association`, the defining ruleset classes, and the rule classes themselves.

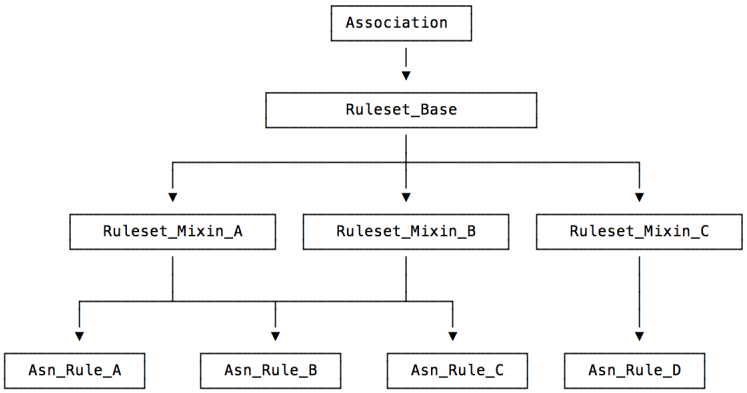


Fig. 9: Rule Inheritance

Where Rules Live: The AssociationRegistry

In order to be used, rules are loaded into an *Association Registry*. The registry is used by the *generate()* to produce the associations. The registry is also used by the *load_asn()* function to validate a potential association data against list of rules.

Association Registry

The *AssociationRegistry* is the rule organizer. An AssociationRegistry is instantiated with the files containing the desired rules. The *match()* method is used to find associations that a member belongs to.

AssociationRegistry is a subclass of py3:dict and supports all of its methods. In particular, multiple AssociationRegistry's can be combined using the update() method.

Association Pool

Association pools are simply tables. Pools are instantiated using the *AssociationPool*. This class is simply a subclass of astropy Table (http://docs.astropy.org/en/stable/table/index.html). As such, any file that is supported by astropy I/O can be used as an association pool.

Each row of a pool defines a member, and the columns define the attributes of that member. It is these attributes that the generator uses to determine which members go into which associations.

Regardless of any implied or explicit typing of data by a table file, internally all data are converted to lowercase strings. It is left up to the individual association definitions on how they will use these attributes.

For JWST Level2/Level3 associations, there is a special case. If an attribute has a value that is equivalent to a Python list:

```
[element, ...]
```

the list will be expanded by the Level2/Level3 associations. This expansion is explained in *Member Attributes that are Lists*

Reference

Association Commands

asn_generate

Association generation is done either using the command line tool asn_generate or through the Python API using either *Main* or *generate()*.

Command Line

```
asn_generate --help
```

Association Candidates

A full explanation of association candidates be found under the *design* section.

Default Rules

The default rules are the *Level2* and *Level3*. Unless the `--ignore-default` option is specified, these rules are included regardless of any other rules also specified by the `-r` options.

DMS Workflow

The JWST pipeline environment has specific requirements that must be met by any task running in that environment. The `--DMS` option ensures that `asn_generate` conforms to those specifications.

API

There are two programmatic entry points: the *Main* class and the *generate()* function. *Main* is the highest level entry and is what is instantiated when the command line `asn_generate` is used. *Main* parses the command line options, creates the *AssociationPool* and *AssociationRegistry* instances, calls `generate`, and saves the resulting associations.

generate() is the main mid-level entry point. Given an *AssociationPool* and an *AssociationRegistry*, *generate()* returns a list of associations.

asn_from_list

Create an association using either the command line tool `asn_from_list` or through the Python API using either `jwst.associations.asn_from_list.Main` or *jwst.associations.asn_from_list.asn_from_list()*

Command Line

```
asn_from_list --help
```

Usage

Level2 Associations

Refer to *Stage 2 Associations: Technical Specifications* for a full description of Level2 associations.

To create a Level2 association, use the following command:

```
asn_from_list -o l2_asn.json -r DMSLevel2bBase *.fits
```

The `-o` option defines the name of the association file to create.

The `-r DMSLevel2bBase` option indicates that a Level2 association is to be created.

Each file in the list will have its own `product` in the association file. When used as input to *calwebb_image2* or *calwebb_spec2*, each product is processed independently, producing the Level2b result for each product.

For those exposures that require an off-target background or imprint image, modify the `members` list for those exposure, adding a new member with an `exptype` of `background` or `imprint` as appropriate. The `expname` for these members are the Level2a exposures that are the background/imprint to use.

An example product that has both a background and imprint exposure would look like the following:

```
"products": [
    {
        "name": "jw99999001001_011001_00001_nirspec",
        "members": [
            {
                "expname": "jw999999001001_011001_00001_nirspec_rate.fits",
                "exptype": "science"
            },
            {
                "expname": "jw999999001001_011001_00002_nirspec_rate.fits",
                "exptype": "background"
            },
            {
                "expname": "jw999999001001_011001_00003_nirspec_rate.fits",
                "exptype": "imprint"
            }
        ]
    }
]
```

Level3 Associations

Refer to *Stage 3 Associations: Technical Specifications* for a full description of Level3 associations.

To create a Level3 association, use the following command:

```
asn_from_list -o l3_asn.json --product-name l3_results *.fits
```

The `-o` option defines the name of the association file to create.

The `--product-name` will set the `name` field that the Level3 calibration code will use as the output name. For the above example, the output files created by *calwebb_image3*, or other Level3 pipelines, will all begin with **l3_results**.

The list of files will all become `science` members of the association, with the presumption that all files will be combined.

For coronagraphic or AMI processing, set the `exptype` of the exposures that are the PSF reference exposures to **psf**. If the PSF files are not in the `members` list, edit the association and add them as members. An example product with a psf exposure would look like:

```
"products": [
    {
        "name": "jw99999-o001_t14_nircam_f182m-mask210r",
        "members": [
            {
                "expname": "jw999999001001_011001_00001_nircam_cal.fits",
                "exptype": "science"
            },
            {
                "expname": "jw999999001001_011001_00002_nircam_cal.fits",
                "exptype": "science"
            },
            {
                "expname": "jw999999001001_011001_00003_nircam_cal.fits",
```

(continues on next page)

(continued from previous page)

```
                "exptype": "psf"
            }
        ]
    }
]
```

API

There are two programmatic entry points: The `Main` is the highest level entry and is what is instantiated when the command line `asn_from_list` is used. `Main` handles the command line interface.

asn_from_list() is the main mid-level entry point.

jwst.associations.asn_from_list Module

Create an association from a list

Functions

asn_from_list(items[, rule])	Creat an association from a list

asn_from_list

jwst.associations.asn_from_list.**asn_from_list**(*items*, *rule=<class 'jwst.associations.lib.rules_level3_base.DMS_Level3_Base'>*, ***kwargs*)

Creat an association from a list

Parameters

- **items** ([*object* (https://docs.python.org/3/library/functions.html#object) *[, ...]*]) – List of items to add.
- **rule** (`Association` rule) – The association rule to use.
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Other named parameters required or pertinent to adding the items to the association.

Returns

association – The association with the items added.

Return type

`Association`-based instance

Notes

This is a lower-level tool for artificially creating an association. As such, the association created may not be valid. It is presume the user knows what they are doing.

asn_gather

Copy members of an association from one location to another.

The association is copied into the destination, re-written such that the member list points to the new location of the members.

Command Line

```
asn_gather --help
```

API

jwst.associations.asn_gather Module

asn_gather: Copy data that is listed in an association

Functions

asn_gather(association[, destination, ...])	Copy members of an association from one location to another

asn_gather

jwst.associations.asn_gather.**asn_gather**(*association*, *destination=None*, *exp_types=None*, *exclude_types=None*, *source_folder=None*, *shellcmd='rsync -urv --no-perms --chmod=ugo=rwX'*)

Copy members of an association from one location to another

The association is copied into the destination, re-written such that the member list points to the new location of the members.

Parameters

- **association** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *pathlib.Path* (https://docs.python.org/3/library/pathlib.html#pathlib.Path)) – The association to gather.
- **destination** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *pathlib.Path* (https://docs.python.org/3/library/pathlib.html#pathlib.Path), *or None*) – The folder to place the association and its members. If None, the current working directory is used.
- **exp_types** (*[str* (https://docs.python.org/3/library/stdtypes.html#str)*[,...]] or None*) – List of exposure types to gather. If None, all are gathered.

- **exclude_types** ([*str* (https://docs.python.org/3/library/stdtypes.html#str)*[,...]] or None*) – List of exposure types to exclude.
- **source_folder** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – Folder where the members originate from. If None, the folder of the association is presumed.
- **shellcmd** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The shell command to use to do the copying of the individual members.

Returns
dest_asn – The copied association.

Return type
pathlib.Path (https://docs.python.org/3/library/pathlib.html#pathlib.Path)

asn_make_pool

Association pool creation from a list of FITS files can be done either using the command line tool `asn_make_pool` or through the Python API *mkpool()*.

Command Line

```
asn_make_pool --help
```

API

jwst.associations.mkpool Module

Tools for pool creation

Functions

mkpool(data[, asn_candidate, dms_note, ...])	Create an association pool from a list of FITS files.

mkpool

jwst.associations.mkpool.**mkpool**(*data*, *asn_candidate=None*, *dms_note=''*, *is_imprt='f'*, *pntgtype='science'*, ***kwargs*)

Create an association pool from a list of FITS files.

Normally, association pools and the associations generated from those pools are created by the automatic ground system process. Users should download and modify those pools if need be. If desired, this function can be used to create pools from scratch using a list of FITS files. Once created, the *generate()* can be used to create associations from these pools.

A number of pool columns used by the Association rules cannot be derived from the header keywords. The columns, and typical other values, are as follows:

•asn_candidate
The observation candidate is always defined in table creation, based on the observation id of each exposure.

However, higher level associations can be created by specifying a list of candidate definitions. An example of adding both background and coronographic candidates would be: [('c1000', 'background'), ('c1001', 'coronographic')]

The specification can be either as a list of 2-tuples, as presented above, or as a single string representation of the list. Using the previous example, the following is also a valid input: "[('c1000', 'background'), ('c1001', 'coronographic')]"

•dms_note
Notes from upstream processing of the downlinked data that may be pertinent to the quality of the data. Currently the value "wfsc_los_jitter" is used by the Level 2 wavefront sensing rule, Asn_Lv2WFSC, to ignore exposures.

•is_imprt
A 't' indicates the exposure is a NIRSpec imprint exposure.

•pntgtype
The general class of exposure. The default value is "science". For target acquisition, the value is "target_acquisition".

Parameters

- **data** (*int* (https://docs.python.org/3/library/functions.html#int)) – The data to get the pool parameters from. Can be pathnames or `astropy.io.fits.HDUL` or `astropy.io.fits.ImageHDU`.
- **asn_candidate** (*[(id, type* (https://docs.python.org/3/library/functions.html#type)*)[, ...]] or None*) – Association candidates to add to each exposure. These are added to the default ('oXXX', 'observation') candidate created from header information.
- **dms_note** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Value for the dms_note column.
- **is_imprt** (*'t' or 'f'*) – Indicator whether exposures are imprint/leakcal exposures.
- **pntgtype** (*'science', 'target_acquisition'*) – General exposure type.
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Other keyword arguments to pass to the `astropy.io.fits.getheader` call.

Returns
pool – The association pool.

Return type
jwst.associations.AssociationPool

Association Rules

Association definitions, or `rules`, are Python classes, all based on *Association*. The base class provides only a framework, much like an abstract base class; all functionality must be implemented in sub-classes.

Any subclass that is intended to produce an association is referred to as a `rule`. Any rule subclass must have a name that begins with the string `Asn_`. This is to ensure that any other classes involved in defining the definition of the rule classes do not get used as rules themselves, such as the *Association* itself.

Association Dynamic Definition

Associations are created by matching members to rules. However, an important concept to remember is that an association is defined by both the rule matched, and by the initial member that matched it. The following example will illustrate this concept.

For JWST *Level 3*, many associations created must have members that all share the same filter. To avoid writing rules for each filter, the rules have a condition that states that it doesn't matter what filter is specified, as long as the association contains all the same filter.

To accomplish this, the association defines a constraint where filter must have a valid value, but can be any valid value. When the association is first attempted to be instantiated with a member, and that member has a valid filter, the association is created. However, the constraint on filter value in the newly created association is modified to match exactly the filter value that the first member had. Now, when other members are attempted to be added to the association, the filter of the new members must match exactly with what the association is expecting.

This dynamic definition allows rules to be written where each value of a specific attribute of a member does not have to be explicitly stated. This provides for very robust, yet concise, set of rule definitions.

User-level API

Core Keys

To be repetitive, the basic association is simply a dict (default) or list. The structure of the dict is completely determined by the rules. However, the base class defines the following keys:

asn_type
: The type of the association.

asn_rule
: The name of the rule.

version_id
: A version number for any associations created by this rule.

code_version
: The version of the generator library in use.

These keys are accessed in the same way any dict key is accessed:

```
asn = Asn_MyAssociation()
print(asn['asn_rule'])

#--> MyAssociation
```

Core Methods

These are the methods of an association rule deal with creation or returning the created association. A rule may define other methods, but the following are required to be implemented.

create()
: Create an association.

add()
: Add a member to the current association.

dump()
: Return the string serialization of the association.

load()
: Return the association from its serialization.

Creation

To create an association based on a member, the `create` method of the rule is called:

```
(association, reprocess_list) = Asn_SomeRule.create(member)
```

`create` returns a 2-tuple: The first element is the association and the second element is a list of `reprocess` instances.

If the member matches the conditions for the rule, an association is returned. If the member does not belong, `None` is returned for the association.

Whether an association is created or not, it is possible a list of `reprocess` instances may be returned. This list represents the expansion of the pool in *Member Attributes that are Lists*

Addition

To add members to an existing association, one uses the *Association.add* method:

```
(matches, reprocess_list) = association.add(new_member)
```

If the association accepts the member, the `matches` element of the 2-tuple will be `True`.

Typically, one does not deal with a single rule, but a collection of rules. For association creation, one typically uses an *AssociationRegistry* to collect all the rules a pool will be compared against. Association registries provide extra functionality to deal with a large and varied set of association rules.

Saving and Loading

Once created, an association can be serialized using its *Association.dump* method. Serialization creates a string representation of the association which can then be saved as one wishes. Some code that does a basic save looks like:

```
file_name, serialized = association.dump()
with open(file_name, 'w') as file_handle:
    file_handle.write(serialized)
```

Note that dump returns a 2-tuple. The first element is the suggested file name to use to save the association. The second element is the serialization.

To retrieve an association, one uses the *Association.load* method:

```
with open(file_name, 'r') as file_handle:
    association = Association.load(file_handle)
```

Association.load will only validate the incoming data against whatever schema or other validation checks the particular subclass calls for. The generally preferred method for loading an association is through the *jwst.associations.load_asn()* function.

Defining New Associations

All association rules are based on the *Association* base class. This class will not create associations on its own; subclasses must be defined. What an association is and how it is later used is completely left to the subclasses. The base class itself only defines the framework required to create associations. The rest of this section will discuss the minimum functionality that a subclass needs to implement in order to create an association.

Class Naming

The *AssociationRegistry* is used to store the association rules. Since rules are defined by Python classes, a way of indicating what the final rule classes are is needed. By definition, rule classes are classes that begin with the string `Asn_`. Only these classes are used to produce associations.

Core Attributes

Since rule classes will potentially have a large number of attributes and methods, the base *Association* class defines two attributes: `data`, which contains the actual association, and `meta`, the structure that holds auxiliary information needed for association creation. Subclasses may redefine these attributes as they see fit. However, it is suggested that they be used as conceptually defined here.

`data` Attribute

`data` contains the association itself. Currently, the base class predefines `data` as a dict. The base class itself is a subclass of `MutableMapping`. Any instance behaves as a dict. The contents of that dict is the contents of the `data` attribute. For example:

```
asn = Asn_MyAssociation()
asn.data['value'] = 'a value'

assert asn['value'] == 'a value'
# True

asn['value'] = 'another value'
assert asn.data['value'] == 'another value'
# True
```

Instantiation

Instantiating a rule, in and of itself, does nothing more than setup the constraints that define the rule, and basic structure initialization.

Implementing `create()`

The base class function performs the following steps:

- Instantiates an instance of the rule
- Calls `add()` to attempt to add the member to the instance

If `add()` returns `matches==False`, then `create` returns `None` as the new association.

Any override of this method is expected to first call `super`. On success, any further initialization may be performed.

Implementing `add()`

The *add()* method adds members to an association.

If a member does belong to the association, the following events occur:

Constraint Modification
: Any wildcard constraints are modified so that any further matching must match exactly the value provided by the current member.

`self._init_hook()` is executed
: If a new association is being created, the rule's `_init_hook` method is executed, if defined. This allows a rule to do further initialization before the member is officially added to the association.

`self._add()` is executed
: The rule class must define `_add()`. This method officially adds the member to the association.

Implementing `dump()` and `load()`

The base `Association` class defines the *dump()* and *load()* methods to serialize the data structure pointing to by the `data` attribute. If the new rule uses the `data` attribute for storing the association information, no further overriding of these methods is necessary.

However, if the new rule does not define `data`, then these methods will need be overridden.

Rule Registration

In order for a rule to be used by `generate`, the rule must be loaded into an `AssociationRegistry`. Since a rule is just a class that is defined as part of a, most likely, larger module, the registry needs to know what classes are rules. Classes to be used as rules are marked with the `RegistryMarker.rule` decorator as follows:

```
# myrules.py
from jwst.associations import (Association, RegistryMarker)

@RegistryMarker.rule
class MyRule(Association):
    ...
```

Then, when the rule file is used to create an `AssociationRegistry`, the class `MyRule` will be included as one of the available rules:

```
>>> from jwst.associations import AssociationRegistry
>>> registry = AssociationRegistry('myrules.py', include_default=False)
>>> print(registry)
    {'MyRule': <class 'abc.MyRule'>}
```

jwst.associations Package

Association Generator

The Association Generator takes a list of items, an Association Pool, and creates sub-lists of those items depending on each item's attributes. How the sub-lists are created is defined by Association Rules.

For more, see the *documentation overview*.

Functions

generate(pool, rules[, version_id, finalize])	Generate associations in the pool according to the rules.
libpath(filepath)	Return the full path to the module library.
load_asn(serialized[, format, first, ...])	Load an Association from a file or object
main([args, pool])	Command-line entrypoint for the association generator

generate

jwst.associations.**generate**(*pool*, *rules*, *version_id=None*, *finalize=True*)

Generate associations in the pool according to the rules.

Parameters

- **pool** (AssociationPool) – The pool to generate from.
- **rules** (AssociationRegistry) – The association rule set.
- **version_id** (*None, True, or str* (https://docs.python.org/3/library/stdtypes.html#str)) – The string to use to tag associations and products. If None, no tagging occurs. If True, use a timestamp If a string, the string.
- **finalize** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Run all rule methods marked as 'finalized'.

Returns

associations – List of associations

Return type

[*Association*[,...]]

Notes

Refer to the *Association Generator* documentation for a full description.

libpath

jwst.associations.**libpath**(*filepath*)

Return the full path to the module library.

load_asn

jwst.associations.**load_asn**(*serialized*, *format=None*, *first=True*, *validate=True*, *registry=<class 'jwst.associations.registry.AssociationRegistry'>*, ***kwargs*)

Load an Association from a file or object

Parameters

- **serialized** (*object* (https://docs.python.org/3/library/functions.html#object)) – The serialized form of the association.
- **format** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – The format to force. If None, try all available.
- **validate** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Validate against the class' defined schema, if any.
- **first** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – A serialization potentially matches many rules. Only return the first succesful load.
- **registry** (AssociationRegistry *or None*) – The *AssociationRegistry* to use. If None, no registry is used. Can be passed just a registry class instead of instance.
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Other arguments to pass to the `load` methods defined in the `Association.IORegistry`

Return type

The Association object

Raises

AssociationNotValidError – Cannot create or validate the association.

Notes

The `serialized` object can be in any format supported by the registered I/O routines. For example, for `json` (https://docs.python.org/3/library/json.html#module-json) and `yaml` formats, the input can be either a string or a file object containing the string.

If no registry is specified, the default *Association.load* method is used.

main

jwst.associations.**main**(*args=None*, *pool=None*)

Command-line entrypoint for the association generator

Wrapper around *Main.cli* so that the return is either True or an exception.

Parameters

- **args** (*[str* (https://docs.python.org/3/library/stdtypes.html#str), *...], or None*) – The command line arguments. Can be one of
  - None (https://docs.python.org/3/library/constants.html#None): sys.argv (https://docs.python.org/3/library/sys.html#sys.argv) is then used.
  - [str, ...]: A list of strings which create the command line with the similar structure as sys.argv (https://docs.python.org/3/library/sys.html#sys.argv)
- **pool** (*None or AssociationPool*) – If None (https://docs.python.org/3/library/constants.html#None), a pool file must be specified in the args. Otherwise, an *AssociationPool*

Classes

Association([version_id])	Association Base Class
AssociationError	Basic errors related to Associations
AssociationNotAConstraint	No matching constraint found
AssociationNotValidError	Given data structure is not a valid association
AssociationPool(*args, **kwargs)	Association Pool
AssociationRegistry([definition_files, ...])	The available associations
ListCategory(value[, names, module, ...])	
Main([args, pool])	Generate Associations from an Association Pool
ProcessItem(obj)	Items to be processed
ProcessList([items, rules, work_over, ...])	A Process list
ProcessQueue	Make a deque iterable and mutable
ProcessQueueSorted([init])	Sort ProcessItem based on work_over
RegistryMarker()	Mark rules, callbacks, and modules for inclusion into a registry

Association

class jwst.associations.**Association**(*version_id=None*)

Bases: MutableMapping (https://docs.python.org/3/library/collections.abc.html#collections.abc.MutableMapping)

Association Base Class

Parameters

version_id (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – Version ID to use in the name of this association. If None, nothing is added.

Raises

AssociationError – If an item doesn't match.

instance

The instance is the association data structure. See *data* below

Type
dict-like

meta

Information about the association.

Type
dict (https://docs.python.org/3/library/stdtypes.html#dict)

data

The association. The format of this data structure is determined by the individual associations and, if defined, validated against their specified schema.

Type
dict (https://docs.python.org/3/library/stdtypes.html#dict)

schema_file

The name of the output schema that an association must adhere to.

Type
str (https://docs.python.org/3/library/stdtypes.html#str)

Attributes Summary

DEFAULT_EVALUATE	Default do not evaluate input values
DEFAULT_FORCE_UNIQUE	Default whether to force constraints to use unique values.
DEFAULT_REQUIRE_CONSTRAINT	Default require that the constraint exists or otherwise can be explicitly checked.
GLOBAL_CONSTRAINT	Global constraints
INVALID_VALUES	Attribute values that indicate the attribute is not specified.
asn_name	Suggest filename for the association
asn_rule	Name of the rule
ioregistry	The association IO registry
is_valid	Check if association is valid
registry	Registry this rule has been placed in.

Methods Summary

add(item[, check_constraints])	Add the item to the association
check_and_set_constraints(item)	Check whether the given dictionaries match parameters for for this association
create(item[, version_id])	Create association if item belongs
dump([format])	Serialize the association
finalize()	Finalize association
is_item_member(item)	Check if item is already a member of this association
items()	
keys()	
load(serialized[, format, validate])	Marshall a previously serialized association
match_constraint(item, constraint, conditions)	Generic constraint checking
validate(asn)	Validate an association against this rule
values()	

Attributes Documentation

DEFAULT_EVALUATE = False

Default do not evaluate input values

DEFAULT_FORCE_UNIQUE = False

Default whether to force constraints to use unique values.

DEFAULT_REQUIRE_CONSTRAINT = True

Default require that the constraint exists or otherwise can be explicitly checked.

GLOBAL_CONSTRAINT = None

Global constraints

INVALID_VALUES = None

Attribute values that indicate the attribute is not specified.

asn_name

Suggest filename for the association

asn_rule

Name of the rule

ioregistry = {'json': <class 'jwst.associations.association_io.json'>, 'yaml': <class 'jwst.associations.association_io.yaml'>}

The association IO registry

is_valid

Check if association is valid

registry = None

Registry this rule has been placed in.

Methods Documentation

add(*item*, *check_constraints=True*)

Add the item to the association

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to add.
- **check_constraints** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If True, see if the item should belong to this association. If False, just add it.

Returns

2-tuple consisting of:

- bool : True if match
- [ProcessList[, ...]]: List of items to process again.

Return type
(match, reprocess_list)

check_and_set_constraints(*item*)

Check whether the given dictionaries match parameters for for this association

Parameters
item (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The parameters to check/set for this association. This can be a list of dictionaries.

Returns

2-tuple consisting of:

- bool : Did constraint match?
- [ProcessItem[, ...]]: List of items to process again.

Return type
(match, reprocess)

classmethod create(*item*, *version_id=None*)

Create association if item belongs

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to initialize the association with.
- **version_id** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – Version ID to use in the name of this association. If None, nothing is added.

Returns

2-tuple consisting of:

- association or None: The association or, if the item does not match this rule, None
- [ProcessList[, ...]]: List of items to process again.

Return type
(association, reprocess_list)

dump(*format='json'*, ***kwargs*)

Serialize the association

Parameters

- **format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format to use to dump the association into.
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – List of arguments to pass to the registered routines for the current association type.

Returns

Tuple where the first item is the suggested base name for the file. Second item is the serialization.

Return type

(name, serialized)

Raises

- ***AssociationError*** – If the operation cannot be done
- ***AssociationNotValidError*** – If the given association does not validate.

finalize()

Finalize association

Finalize or close-off this association. Perform validations, modifications, etc. to ensure that the association is complete.

Returns

associations – List of fully-qualified associations that this association represents. None (https://docs.python.org/3/library/constants.html#None) if a complete association cannot be produced.

Return type

[association[, ...]] or None

is_item_member(*item*)

Check if item is already a member of this association

Parameters

item (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to add.

Returns

is_item_member – True if item is a member.

Return type

bool (https://docs.python.org/3/library/functions.html#bool)

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

classmethod load(*serialized*, *format=None*, *validate=True*, ***kwargs*)

Marshall a previously serialized association

Parameters

- **serialized** (*object* (https://docs.python.org/3/library/functions.html#object)) – The serialized form of the association.
- **format** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – The format to force. If None, try all available.

- **validate** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Validate against the class' defined schema, if any.
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Other arguments to pass to the *load* method

Returns
association – The association.

Return type
Association

Raises
AssociationNotValidError – Cannot create or validate the association.

Notes

The `serialized` object can be in any format supported by the registered I/O routines. For example, for `json` (https://docs.python.org/3/library/json.html#module-json) and `yaml` formats, the input can be either a string or a file object containing the string.

match_constraint(*item*, *constraint*, *conditions*)

Generic constraint checking

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to retrieve the values from
- **constraint** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The name of the constraint
- **conditions** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The conditions structure

Returns

2-tuple consisting of:

- bool : True if the all constraints are satisfied
- [ProcessList[, ...]]: List of items to process again.

Return type
(matches, reprocess_list)

classmethod validate(*asn*)

Validate an association against this rule

Parameters
asn (*Association or association-like*) – The association structure to examine

Returns
valid – True if valid. Otherwise the *AssociationNotValidError* is raised

Return type
bool (https://docs.python.org/3/library/functions.html#bool)

Raises
AssociationNotValidError – If there is some reason validation failed.

Notes

The base method checks against the rule class' schema If the rule class does not define a schema, a warning is issued but the routine will return True.

values() → an object providing a view on D's values

AssociationError

exception jwst.associations.**AssociationError**

Basic errors related to Associations

AssociationNotAConstraint

exception jwst.associations.**AssociationNotAConstraint**

No matching constraint found

AssociationNotValidError

exception jwst.associations.**AssociationNotValidError**

Given data structure is not a valid association

AssociationPool

class jwst.associations.**AssociationPool**(**args*, ***kwargs*)

Bases: Table

Association Pool

An AssociationPool is essentially an astropy Table with the following default behaviors:

- ASCII tables with a default delimiter of |
- All values are read in as strings

Methods Summary

read(filename[, delimiter, format])	Read in a Pool file
write(*args, **kwargs)	Write the pool to a file.

Methods Documentation

classmethod read(*filename*, *delimiter='|'*, *format='ascii'*, ***kwargs*)

Read in a Pool file

Parameters

- **filename** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – File path to read in as a table.
- **delimiter** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Character used to delineate columns.
- **format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format of the input file.

Returns

The `AssociationPool` representation of the file.

Return type

AssociationPool

write(**args*, ***kwargs*)

Write the pool to a file.

Parameters

- **output** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *file-like*) – The output file or file-like object.
- **delimiter** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The string to use to delineate columns. Default is '|'.
- **format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format the file should be written in. Default is 'ascii'.
- **args** (*obj*) – Other parameters that `astropy.io.ascii.write` can accept.
- **kwargs** (*obj*) – Other parameters that `astropy.io.ascii.write` can accept.

AssociationRegistry

class jwst.associations.**AssociationRegistry**(*definition_files=None*, *include_default=True*, *global_constraints=None*, *name=None*, *include_bases=False*)

Bases: `dict` (https://docs.python.org/3/library/stdtypes.html#dict)

The available associations

Parameters

- **definition_files** (*[str* (https://docs.python.org/3/library/stdtypes.html#str)*,]*) – The files to find the association definitions in.
- **include_default** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – True to include the default definitions.
- **global_constraints** (`Constraint`) – Constraints to be added to each rule.
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – An identifying string, used to prefix rule names.

- **include_bases** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If True, include base classes not considered rules.

Notes

The general workflow is as follows:

•Create the registry

```
>>> from jwst.associations.registry import AssociationRegistry
>>> registry = AssociationRegistry()
```

•Create associations from an item

```
>>> associations, reprocess = registry.match(item)
```

•Finalize the associations

```
>>> final_asns = registry.callback.reduce('finalize', associations)
```

In practice, this is one step in a larger loop over all items to be associated. This does not account for adding items to already existing associations. See *generate()* for more information.

Attributes Summary

rule_set	Rules within the Registry

Methods Summary

add_rule(name, obj[, global_constraints])	Add object as rule to registry
load(serialized[, format, validate, first])	Load a previously serialized association
match(item[, version_id, allow, ignore])	See if item belongs to any of the associations defined.
populate(module[, global_constraints, ...])	Parse out all rules and callbacks in a module and add them to the registry
validate(association)	Validate a given association

Attributes Documentation

rule_set

Rules within the Registry

Methods Documentation

add_rule(*name*, *obj*, *global_constraints=None*)

Add object as rule to registry

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Name of the object
- **obj** (*object* (https://docs.python.org/3/library/functions.html#object)) – The object to be considered a rule
- **global_constraints** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The global constraints to attach to the rule.

load(*serialized*, *format=None*, *validate=True*, *first=True*, ***kwargs*)

Load a previously serialized association

Parameters

- **serialized** (*object* (https://docs.python.org/3/library/functions.html#object)) – The serialized form of the association.
- **format** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – The format to force. If None, try all available.
- **validate** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Validate against the class' defined schema, if any.
- **first** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – A serialization potentially matches many rules. Only return the first succesful load.
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Other arguments to pass to the *load* method

Return type

The Association object, or the list of association objects.

Raises

AssociationError – Cannot create or validate the association.

match(*item*, *version_id=None*, *allow=None*, *ignore=None*)

See if item belongs to any of the associations defined.

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – An item, like from a Pool, to find associations for.
- **version_id** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – If specified, a string appended to association names. If None, nothing is used.
- **allow** (*[type* (https://docs.python.org/3/library/functions.html#type)*(Association), ...]*) – List of rules to allow to be matched. If None, all available rules will be used.
- **ignore** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – A list of associations to ignore when looking for a match. Intended to ensure that already created associations are not re-created.

Returns

(associations, reprocess_list) –

associations
[[association,...]] List of associations item belongs to. Empty if none match

reprocess_list
[[AssociationReprocess, ...]] List of reprocess events.

Return type
2-tuple

populate(*module*, *global_constraints=None*, *include_bases=None*)

Parse out all rules and callbacks in a module and add them to the registry

Parameters
module (*module*) – The module, and all submodules, to be parsed.

validate(*association*)

Validate a given association

Parameters
association (*association-like*) – The data to validate

Returns
rules – List of rules that validated

Return type
list (https://docs.python.org/3/library/stdtypes.html#list)

Raises
AssociationNotValidError – Association did not validate

ListCategory

class jwst.associations.**ListCategory**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum (https://docs.python.org/3/library/enum.html#enum.Enum)

Attributes Summary

BOTH
EXISTING
NONSCIENCE
RULES

Attributes Documentation

BOTH = 1

EXISTING = 2

NONSCIENCE = 3

RULES = 0

Main

class jwst.associations.**Main**(*args=None*, *pool=None*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Generate Associations from an Association Pool

Parameters

- **args** (*[str (https://docs.python.org/3/library/stdtypes.html#str), ...], or None*) – The command line arguments. Can be one of
  - None (https://docs.python.org/3/library/constants.html#None): sys.argv (https://docs.python.org/3/library/sys.html#sys.argv) is then used.
  - [str, ...]: A list of strings which create the command line with the similar structure as sys.argv (https://docs.python.org/3/library/sys.html#sys.argv)
- **pool** (*None or AssociationPool*) – If None (https://docs.python.org/3/library/constants.html#None), a pool file must be specified in the args. Otherwise, an *AssociationPool*

pool

The pool read in, or passed in through the parameter *pool*

Type
AssociationPool

rules

The rules used for association creation.

Type
AssociationRegistry

associations

The list of generated associations.

Type
[*Association*, ...]

Notes

Refer to the *Association Generator* documentation for a full description.

Attributes Summary

orphaned	The pool of exposures that do not belong to any association.

Methods Summary

cli([args, pool])	Run the full association generation process
configure([args, pool])	Configure to prepare for generation
generate()	Generate the associations
parse_args([args, has_pool])	Set command line arguments
save()	Save the associations to disk.

Attributes Documentation

orphaned

The pool of exposures that do not belong to any association.

Methods Documentation

classmethod cli(*args=None*, *pool=None*)

Run the full association generation process

Parameters

- **args** (*[str* (https://docs.python.org/3/library/stdtypes.html#str), *...], or None*) – The command line arguments. Can be one of
  - None (https://docs.python.org/3/library/constants.html#None): sys.argv (https://docs.python.org/3/library/sys.html#sys.argv) is then used.
  - [str, ...]: A list of strings which create the command line with the similar structure as sys.argv (https://docs.python.org/3/library/sys.html#sys.argv)
- **pool** (*None or* AssociationPool) – If None (https://docs.python.org/3/library/constants.html#None), a pool file must be specified in the args. Otherwise, an *AssociationPool*

Returns

generator – A fully executed association generator.

Return type

Main

configure(*args=None*, *pool=None*)

Configure to prepare for generation

Parameters

- **args** (*[str (https://docs.python.org/3/library/stdtypes.html#str), ..], or None*) – The command line arguments. Can be one of
  - None (https://docs.python.org/3/library/constants.html#None): sys.argv (https://docs.python.org/3/library/sys.html#sys.argv) is then used.
  - [str, ...]: A list of strings which create the command line with the similar structure as sys.argv (https://docs.python.org/3/library/sys.html#sys.argv)
- **pool** (*None or AssociationPool*) – If None (https://docs.python.org/3/library/constants.html#None), a pool file must be specified in the args. Otherwise, an *AssociationPool*

generate()

Generate the associations

parse_args(*args=None*, *has_pool=False*)

Set command line arguments

Parameters

- **args** (*list (https://docs.python.org/3/library/stdtypes.html#list), str (https://docs.python.org/3/library/stdtypes.html#str), or None*) – List of command-line arguments. If a string, spaces seperate the arguments. If None, sys.argv (https://docs.python.org/3/library/sys.html#sys.argv) is used.
- **has_pool** (*bool-like*) – Do not require *pool* from the command line if a pool is already in hand.

save()

Save the associations to disk.

ProcessItem

class jwst.associations.**ProcessItem**(*obj*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Items to be processed

Parameters

obj (*object* (https://docs.python.org/3/library/functions.html#object)) – The object to make a *ProcessItem*. Objects must be equatable.

Methods Summary

to_process_items(iterable)	Iterable to convert a list to ProcessItem's

Methods Documentation

classmethod to_process_items(*iterable*)

Iterable to convert a list to ProcessItem's

Parameters

iterable (*iterable*) – A source of objects to convert

Returns

- *An iterable where the object has been*
- converted to a *ProcessItem*

ProcessList

class jwst.associations.**ProcessList**(*items=None*, *rules=None*, *work_over=ListCategory.BOTH*, *only_on_match=False*, *trigger_constraints=None*, *trigger_rules=None*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

A Process list

Parameters

- **items** (*[item[, ...]]*) – The list of items to process
- **rules** (*[Association[, ...]]*) – List of rules to process the items against.
- **work_over** (*int* (https://docs.python.org/3/library/functions.html#int)) – What the reprocessing should work on: - ProcessList.EXISTING: Only existing associations - ProcessList.RULES: Only on the rules to create new associations - ProcessList.BOTH: Compare to both existing and rules - ProcessList.NONSCIENCE: Only on non-science items
- **only_on_match** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Only use this object if the overall condition is True.
- **trigger_constraints** (*[Constraint[,...]]*) – The constraints that created the ProcessList
- **trigger_rules** (*[Association[,...]]*) – The association rules that created the ProcessList

Attributes Summary

hash	Create a unique hash

Methods Summary

update(process_list[, full])	Update with information from ProcessList

Attributes Documentation

hash

Create a unique hash

Methods Documentation

update(*process_list*, *full=False*)

Update with information from ProcessList

Attributes from `process_list` are added to self's attributes. If `not full`, the attributes `rules`, 'work_over`, and `only_on_match` are not taken.

Note that if `full`, destructive action will occur with respect to `work_over` and `only_on_match`.

Parameters

- **process_list** (ProcessList) – The source process list to absorb.
- **full** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Include the hash attributes `rules`, `work_over`, and `only_on_match`.

ProcessQueue

class jwst.associations.**ProcessQueue**

Bases: deque (https://docs.python.org/3/library/collections.html#collections.deque)

Make a deque iterable and mutable

ProcessQueueSorted

class jwst.associations.**ProcessQueueSorted**(*init=None*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Sort ProcessItem based on work_over

`ProcessList`s are handled in order of `RULES`, `BOTH`, `EXISTING`, and `NONSCIENCE`.

Parameters

init (*[ProcessList[,...]]*) – List of *ProcessList* to start the queue with.

Methods Summary

extend(process_lists)	Add the list of process items to their appropriate queues

Methods Documentation

extend(*process_lists*)

Add the list of process items to their appropriate queues

RegistryMarker

class jwst.associations.**RegistryMarker**

Bases: object (https://docs.python.org/3/library/functions.html#object)

Mark rules, callbacks, and modules for inclusion into a registry

Methods Summary

callback(event)	Mark object as a callback for an event
is_marked(obj)	Has an objected been marked?
mark(obj)	Mark that an object should be part of the registry
rule(obj)	Mark object as rule
schema(filename)	Mark a file as a schema source
utility(class_obj)	Mark the class as a Utility class

Methods Documentation

static callback(*event*)

Mark object as a callback for an event

Parameters

- **event** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Event this is a callback for.
- **obj** (*func*) – Function, or any callable, to be called when the corresponding event is triggered.

Returns

Function to use as a decorator for the object to be marked.

Return type

func

Notes

The following attributes are added to the object:

- **_asnreg_role**
  ['callback'] The role the object as been assigned.
- **_asnreg_events**
  [[event[, ...]]] The events this callable object is a callback for.
- **_asnreg_mark**
  [True] Indicated that the object has been marked.

static is_marked(*obj*)

Has an objected been marked?

static mark(*obj*)

Mark that an object should be part of the registry

Parameters
obj (*object* (https://docs.python.org/3/library/functions.html#object)) – The object to mark

Returns
Object that has been marked. Returned to enable use as a decorator.

Return type
obj

Notes

The following attributes are added to the object:

- **_asnreg_mark**
  [True] Attribute added to object and is set to True
- **_asnreg_role**
  [str or None] If not already assigned, the role is left unspecified using None.

static rule(*obj*)

Mark object as rule

Parameters
obj (*object* (https://docs.python.org/3/library/functions.html#object)) – The object that should be treated as a rule

Returns
obj – Return object to enable use as a decorator.

Return type
object (https://docs.python.org/3/library/functions.html#object)

Notes

The following attributes are added to the object:

- **_asnreg_role**
  ['rule'] Attributed added to object and set to *rule*
- **_asnreg_mark**
  [True] Attributed added to object and set to True

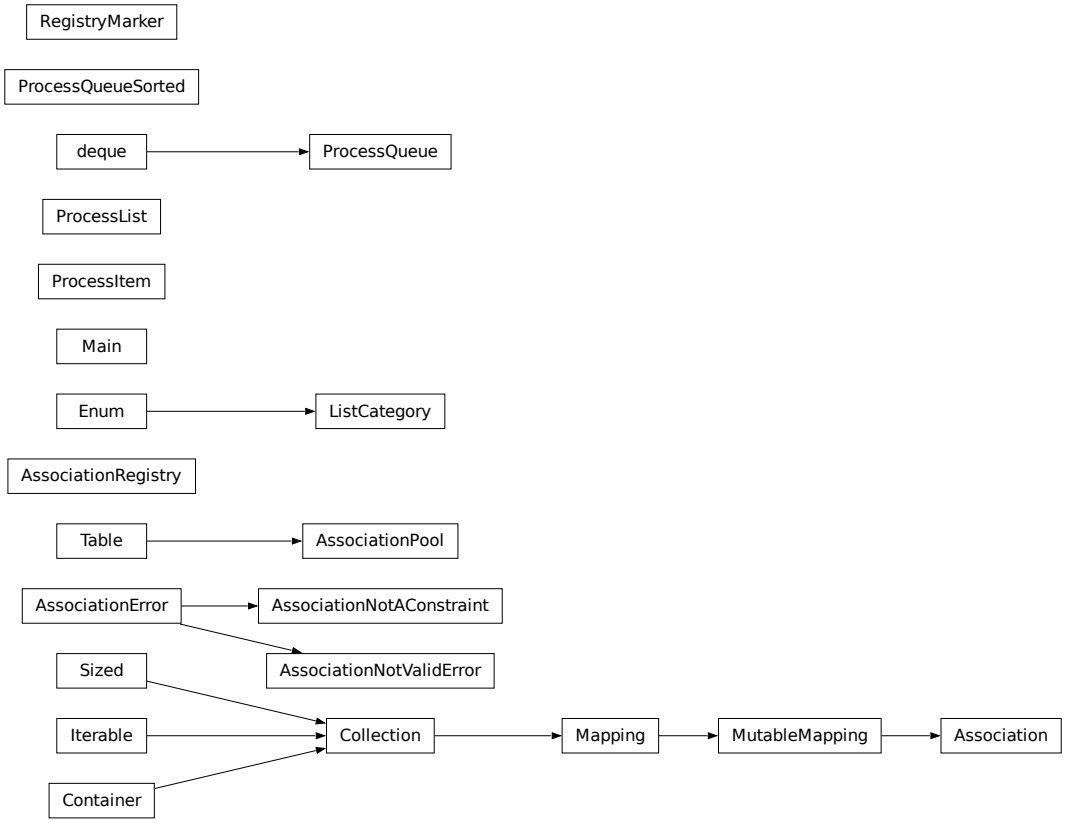
static schema(*filename*)

Mark a file as a schema source

static utility(*class_obj*)

Mark the class as a Utility class

Class Inheritance Diagram



jwst.associations.lib.rules_level2b Module

Association Definitions: DMS Level2b product associations

Classes

Asn_Lv2CoronAsRate(*args, **kwargs)	Create normal rate products for some coronographic data
Asn_Lv2FGS(*args, **kwargs)	Level2b FGS Association
Asn_Lv2Image(*args, **kwargs)	Level2b Non-TSO Science Image Association
Asn_Lv2ImageNonScience(*args, **kwargs)	Level2b Non-science Image Association
Asn_Lv2ImageSpecial(*args, **kwargs)	Level2b Auxiliary Science Image Association
Asn_Lv2ImageTSO(*args, **kwargs)	Level2b Time Series Science Image Association
Asn_Lv2MIRLRSFixedSlitNod(*args, **kwargs)	Level2b MIRI LRS Fixed Slit background nods Association
Asn_Lv2NRSFSS(*args, **kwargs)	Level2b NIRSpec Fixed-slit Association
Asn_Lv2NRSIFUNod(*args, **kwargs)	Level2b NIRSpec IFU Association
Asn_Lv2NRSLAMPImage(*args, **kwargs)	Level2b NIRSpec image Lamp Calibrations Association
Asn_Lv2NRSLAMPSpectral(*args, **kwargs)	Level2b NIRSpec spectral Lamp Calibrations Association
Asn_Lv2NRSMSA(*args, **kwargs)	Level2b NIRSpec MSA Association
Asn_Lv2Spec(*args, **kwargs)	Level2b Science Spectral Association
Asn_Lv2SpecImprint(*args, **kwargs)	Level2b Treat Imprint/Leakcal as science
Asn_Lv2SpecSpecial(*args, **kwargs)	Level2b Auxiliary Science Spectral Association
Asn_Lv2SpecTSO(*args, **kwargs)	Level2b Time Series Science Spectral Association
Asn_Lv2WFSSNIS(*args, **kwargs)	Level2b WFSS/GRISM Association
Asn_Lv2WFSSNRC(*args, **kwargs)	Level2b WFSS/GRISM Association
Asn_Lv2WFSC(*args, **kwargs)	Level2b Wavefront Sensing & Control Association

Asn_Lv2CoronAsRate

class jwst.associations.lib.rules_level2b.**Asn_Lv2CoronAsRate**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Image, DMSLevel2bBase

Create normal rate products for some coronographic data

Characteristics;

- Association type: image2
- Pipeline: calwebb_image2
- NIRCam Coronagraphic
- Only subarray=Full exposures
- Treat as non-timeseries, producing "rate" products

Methods Summary

is_item_coron(item)	Override to always return false

Methods Documentation

is_item_coron(*item*)

Override to always return false

The override will force `make_member` to create a "rate" product instead of a "rateints" product.

Asn_Lv2FGS

class jwst.associations.lib.rules_level2b.**Asn_Lv2FGS**(**args*, ***kwargs*)

Bases: `AsnMixin_Lv2Image`, `DMSLevel2bBase`

Level2b FGS Association

Characteristics:

- Association type: `image2`
- Pipeline: `calwebb_image2`
- Image-based FGS science exposures
- Single science exposure

Asn_Lv2Image

class jwst.associations.lib.rules_level2b.**Asn_Lv2Image**(**args*, ***kwargs*)

Bases: `AsnMixin_Lv2Image`, `DMSLevel2bBase`

Level2b Non-TSO Science Image Association

Characteristics:

- Association type: `image2`
- Pipeline: `calwebb_image2`
- Image-based science exposures
- Single science exposure
- Non-TSO
- Non-coronagraphic

Asn_Lv2ImageNonScience

class jwst.associations.lib.rules_level2b.**Asn_Lv2ImageNonScience**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Special, AsnMixin_Lv2Image, DMSLevel2bBase

Level2b Non-science Image Association

Characteristics:

- Association type: image2
- Pipeline: calwebb_image2
- Image-based non-science exposures, such as target acquisitions
- Single science exposure

Asn_Lv2ImageSpecial

class jwst.associations.lib.rules_level2b.**Asn_Lv2ImageSpecial**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Special, AsnMixin_Lv2Image, DMSLevel2bBase

Level2b Auxiliary Science Image Association

Characteristics:

- Association type: image2
- Pipeline: calwebb_image2
- Image-based science exposures that are to be used as background or PSF exposures
- Single science exposure
- No other exposure can be part of the association

Asn_Lv2ImageTSO

class jwst.associations.lib.rules_level2b.**Asn_Lv2ImageTSO**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Image, DMSLevel2bBase

Level2b Time Series Science Image Association

Characteristics:

- Association type: tso-image2
- Pipeline: calwebb_tso-image2
- Image-based Time Series exposures
- Single science exposure

Asn_Lv2MIRLRSFixedSlitNod

class jwst.associations.lib.rules_level2b.**Asn_Lv2MIRLRSFixedSlitNod**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b MIRI LRS Fixed Slit background nods Association

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- MIRI LRS Fixed slit
- Single science exposure
- Include slit nods as backgrounds

Methods Summary

get_exposure_type(item[, default])	Modify exposure type depending on dither pointing index

Methods Documentation

get_exposure_type(*item*, *default='science'*)

Modify exposure type depending on dither pointing index

Behaves as the superclass method. However, if the constraint is_current_patt_num is True, mark the exposure type as background.

Asn_Lv2NRSFSS

class jwst.associations.lib.rules_level2b.**Asn_Lv2NRSFSS**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Nod, AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b NIRSpec Fixed-slit Association

Notes

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Spectral-based NIRSpec fixed-slit single target science exposures
- Single science exposure
- Handle along-the-slit background nodding

Association includes both the background and science exposures of the nodding. The identified science exposure is fixed by the nod, pattern, and exposure number to prevent other science exposures being included.

Asn_Lv2NRSIFUNod

class jwst.associations.lib.rules_level2b.**Asn_Lv2NRSIFUNod**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Nod, AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b NIRSpec IFU Association

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Spectral-based NIRSpec IFU multi-object science exposures
- Single science exposure
- Handle 2 and 4 point background nodding
- Include related imprint exposures

Asn_Lv2NRSLAMPImage

class jwst.associations.lib.rules_level2b.**Asn_Lv2NRSLAMPImage**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Image, AsnMixin_Lv2Special, DMSLevel2bBase

Level2b NIRSpec image Lamp Calibrations Association

Characteristics:

- Association type: image2
- Pipeline: calwebb_image2
- Image-based calibration exposures
- Single science exposure

Asn_Lv2NRSLAMPSpectral

class jwst.associations.lib.rules_level2b.**Asn_Lv2NRSLAMPSpectral**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Special, DMSLevel2bBase

Level2b NIRSpec spectral Lamp Calibrations Association

Characteristics:

- Association type: nrslamp-spec2
- Pipeline: calwebb_nrslamp-spec2
- Spectral-based calibration exposures
- Single science exposure

Asn_Lv2NRSMSA

class jwst.associations.lib.rules_level2b.**Asn_Lv2NRSMSA**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Nod, AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b NIRSpec MSA Association

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Spectral-based NIRSpec MSA multi-object science exposures
- Single science exposure
- Handle slitlet nodding for background subtraction

Asn_Lv2Spec

class jwst.associations.lib.rules_level2b.**Asn_Lv2Spec**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b Science Spectral Association

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Spectral-based single target science exposures
- Single science exposure
- Non-TSO
- Not part of a background dither observation

Asn_Lv2SpecImprint

class jwst.associations.lib.rules_level2b.**Asn_Lv2SpecImprint**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Special, AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b Treat Imprint/Leakcal as science

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Only handles Imprint/Leakcal exposures
- Single science exposure

Asn_Lv2SpecSpecial

class jwst.associations.lib.rules_level2b.**Asn_Lv2SpecSpecial**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Special, AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b Auxiliary Science Spectral Association

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Spectral-based single target science exposures that are background exposures
- Single science exposure

Asn_Lv2SpecTSO

class jwst.associations.lib.rules_level2b.**Asn_Lv2SpecTSO**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2Spectral, DMSLevel2bBase

Level2b Time Series Science Spectral Association

Characteristics:

- Association type: tso-spec2
- Pipeline: calwebb_tso-spec2
- Spectral-based single target time series exposures
- Single science exposure
- No other exposure can be part of the association

Asn_Lv2WFSSNIS

class jwst.associations.lib.rules_level2b.**Asn_Lv2WFSSNIS**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2WFSS, AsnMixin_Lv2Spectral

Level2b WFSS/GRISM Association

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Multi-object science exposures
- Single science exposure
- Require a source catalog from processing of the corresponding direct imagery.

Asn_Lv2WFSSNRC

class jwst.associations.lib.rules_level2b.**Asn_Lv2WFSSNRC**(**args*, ***kwargs*)

Bases: AsnMixin_Lv2WFSS, AsnMixin_Lv2Spectral

Level2b WFSS/GRISM Association

Characteristics:

- Association type: spec2
- Pipeline: calwebb_spec2
- Multi-object science exposures
- Single science exposure
- Require a source catalog from processing of the corresponding direct imagery.

Asn_Lv2WFSC

class jwst.associations.lib.rules_level2b.**Asn_Lv2WFSC**(**args*, ***kwargs*)

Bases: DMSLevel2bBase

Level2b Wavefront Sensing & Control Association

Characteristics:

- Association type: wfs-image2
- Pipeline: calwebb_wfs-image2
- WFS and WFS&C observations
- Single science exposure

Class Inheritance Diagram

jwst.associations.lib.rules_level3 Module

Association Definitions: DMS Level3 product associations

Classes

Asn_Lv3ACQ_Reprocess(*args, **kwargs)	Level 3 Gather Target Acquisitions
Asn_Lv3AMI(*args, **kwargs)	Level 3 Aperture Mask Interferometry Association
Asn_Lv3Image(*args, **kwargs)	Level 3 Science Image Association
Asn_Lv3ImageBackground(*args, **kwargs)	Level 3 Background Image Association
Asn_Lv3MIRCoron(*args, **kwargs)	Level 3 Coronagraphy Association
Asn_Lv3MIRMRS(*args, **kwargs)	Level 3 MIRI MRS Association
Asn_Lv3MIRMRSBackground(*args, **kwargs)	Level 3 MIRI MRS Association Auxiliary data
Asn_Lv3NRCCoron(*args, **kwargs)	Level 3 Coronagraphy Association
Asn_Lv3NRCCoronImage(*args, **kwargs)	Level 3 Coronagraphy Association handled as regular imaging
Asn_Lv3NRSFSS(*args, **kwargs)	Level 3 NIRSpec Fixed-slit Science
Asn_Lv3NRSIFU(*args, **kwargs)	Level 3 IFU gratings Association
Asn_Lv3NRSIFUBackground(*args, **kwargs)	Level 3 Spectral Association
Asn_Lv3SlitlessSpectral(*args, **kwargs)	Level 3 slitless, target-based or single-object spectrographic Association
Asn_Lv3SpecAux(*args, **kwargs)	Level 3 Spectral Association
Asn_Lv3SpectralSource(*args, **kwargs)	Level 3 slit-like, multi-object spectrographic Association
Asn_Lv3SpectralTarget(*args, **kwargs)	Level 3 slit-like, target-based or single-object spectrographic Association
Asn_Lv3TSO(*args, **kwargs)	Level 3 Time-Series Association
Asn_Lv3WFSCMB(*args, **kwargs)	Level 3 Wavefront Control & Sensing Association
Asn_Lv3WFSSNIS(*args, **kwargs)	Level 3 WFSS/Grism Association

Asn_Lv3ACQ_Reprocess

class jwst.associations.lib.rules_level3.**Asn_Lv3ACQ_Reprocess**(**args*, ***kwargs*)

Bases: DMS_Level3_Base

Level 3 Gather Target Acquisitions

Characteristics:

- Association type: Not applicable
- Pipeline: Not applicable
- Used to populate other related associations

Notes

For first loop, simply send acquisitions and confirms back.

Asn_Lv3AMI

class jwst.associations.lib.rules_level3.**Asn_Lv3AMI**(**args*, ***kwargs*)

Bases: AsnMixin_Science

Level 3 Aperture Mask Interferometry Association

Characteristics:

- Association type: ami3
- Pipeline: calwebb_ami3
- Gather science and related PSF exposures

Notes

AMI is nearly completely defined by the association candidates produced by APT. Tracking Issues:

- github #310 (https://github.com/STScI-JWST/jwst/issues/310)

Asn_Lv3Image

class jwst.associations.lib.rules_level3.**Asn_Lv3Image**(**args*, ***kwargs*)

Bases: AsnMixin_Science

Level 3 Science Image Association

Characteristics:

- Association type: image3
- Pipeline: calwebb_image3
- Non-TSO
- Non-WFS&C

Asn_Lv3ImageBackground

class jwst.associations.lib.rules_level3.**Asn_Lv3ImageBackground**(**args*, ***kwargs*)

Bases: AsnMixin_AuxData, AsnMixin_Science

Level 3 Background Image Association

Characteristics:

- Association type: image3
- Pipeline: calwebb_image3
- Non-TSO
- Non-WFS&C

Asn_Lv3MIRCoron

class jwst.associations.lib.rules_level3.**Asn_Lv3MIRCoron**(**args*, ***kwargs*)

Bases: AsnMixin_Coronagraphy, AsnMixin_Science

Level 3 Coronagraphy Association

Characteristics:

- Association type: coron3
- Pipeline: calwebb_coron3
- MIRI Coronagraphy
- Gather science and related PSF exposures

Notes

Coronagraphy is nearly completely defined by the association candidates produced by APT. Tracking Issues:

- github #311 (https://github.com/STScI-JWST/jwst/issues/311)
- JP-3219 (https://jira.stsci.edu/browse/JP-3219)

Asn_Lv3MIRMRS

class jwst.associations.lib.rules_level3.**Asn_Lv3MIRMRS**(**args*, ***kwargs*)

Bases: AsnMixin_Spectrum

Level 3 MIRI MRS Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- Just MIRI MRS
- optical path determined by calibration
- Cannot be TSO
- Must have pattern type defined

Attributes Summary

dms_product_name	Define product name.

Attributes Documentation

dms_product_name

Asn_Lv3MIRMRSBackground

class jwst.associations.lib.rules_level3.**Asn_Lv3MIRMRSBackground**(**args*, ***kwargs*)

Bases: AsnMixin_AuxData, AsnMixin_Spectrum

Level 3 MIRI MRS Association Auxiliary data

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- Just MIRI MRS
- optical path determined by calibration
- Cannot be TSO
- Must have pattern type defined

Attributes Summary

dms_product_name	Define product name.

Attributes Documentation

dms_product_name

Asn_Lv3NRCCoron

class jwst.associations.lib.rules_level3.**Asn_Lv3NRCCoron**(**args*, ***kwargs*)

Bases: AsnMixin_Coronagraphy, AsnMixin_Science

Level 3 Coronagraphy Association

Characteristics:

- Association type: coron3
- Pipeline: calwebb_coron3
- Gather science and related PSF exposures
- Exclude “extra” NIRCam detectors that don’t have target on them

Notes

Coronagraphy is nearly completely defined by the association candidates produced by APT. Tracking Issues:

- github #311 (https://github.com/STScI-JWST/jwst/issues/311)
- JP-3219 (https://jira.stsci.edu/browse/JP-3219)

Asn_Lv3NRCCoronImage

class jwst.associations.lib.rules_level3.**Asn_Lv3NRCCoronImage**(**args*, ***kwargs*)

Bases: AsnMixin_Science

Level 3 Coronagraphy Association handled as regular imaging

Characteristics:

- Association type: image3
- Pipeline: calwebb_image3
- Gather science exposures only, no psf exposures
- Only include NRC SW images taken in full-frame

Attributes Summary

dms_product_name	Define product name.

Methods Summary

is_item_coron(item)	Override to ignore coronographic designation

Attributes Documentation

dms_product_name

Methods Documentation

is_item_coron(*item*)

Override to ignore coronographic designation

Coronagraphic data is to be processed both as coronagraphic (by default), but also as just plain imaging. Coronagraphic data is processed using the Asn_Lv3Coron rule. This rule will handle the creation of the image version. It causes the input members to be of type "cal", instead of "calints".

Asn_Lv3NRSFSS

class jwst.associations.lib.rules_level3.**Asn_Lv3NRSFSS**(**args*, ***kwargs*)

Bases: AsnMixin_Spectrum

Level 3 NIRSpec Fixed-slit Science

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- NIRSpec Fixed-slit Science
- Non-TSO

Attributes Summary

dms_product_name	Define product name.

Attributes Documentation

dms_product_name

Asn_Lv3NRSIFU

class jwst.associations.lib.rules_level3.**Asn_Lv3NRSIFU**(**args*, ***kwargs*)

Bases: AsnMixin_Spectrum

Level 3 IFU gratings Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- optical path determined by calibration

Asn_Lv3NRSIFUBackground

class jwst.associations.lib.rules_level3.**Asn_Lv3NRSIFUBackground**(**args*, ***kwargs*)

Bases: AsnMixin_AuxData, AsnMixin_Spectrum

Level 3 Spectral Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3

Asn_Lv3SlitlessSpectral

class jwst.associations.lib.rules_level3.**Asn_Lv3SlitlessSpectral**(**args*, ***kwargs*)

Bases: AsnMixin_Spectrum

Level 3 slitless, target-based or single-object spectrographic Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- Single target
- Non-TSO

Asn_Lv3SpecAux

class jwst.associations.lib.rules_level3.**Asn_Lv3SpecAux**(**args*, ***kwargs*)

Bases: AsnMixin_AuxData, AsnMixin_Spectrum

Level 3 Spectral Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3

Asn_Lv3SpectralSource

class jwst.associations.lib.rules_level3.**Asn_Lv3SpectralSource**(**args*, ***kwargs*)

Bases: AsnMixin_Spectrum

Level 3 slit-like, multi-object spectrographic Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- Multi-object
- Non-TSO

Attributes Summary

dms_product_name	Define product name.

Attributes Documentation

dms_product_name

Asn_Lv3SpectralTarget

class jwst.associations.lib.rules_level3.**Asn_Lv3SpectralTarget**(**args*, ***kwargs*)

Bases: AsnMixin_Spectrum

Level 3 slit-like, target-based or single-object spectrographic Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- Single target
- Non-TSO

Methods Summary

finalize()	Finalize association

Methods Documentation

finalize()

Finalize association

For NRS Fixed-slit, finalization means creating new members for the background nods.

Returns

associations – List of fully-qualified associations that this association represents. None (https://docs.python.org/3/library/constants.html#None) if a complete association cannot be produced.

Return type

[association[, ...]] or None

Asn_Lv3TSO

class jwst.associations.lib.rules_level3.**Asn_Lv3TSO**(**args*, ***kwargs*)

Bases: AsnMixin_Science

Level 3 Time-Series Association

Characteristics:

- Association type: tso3
- Pipeline: calwebb_tso3

Asn_Lv3WFSCMB

class jwst.associations.lib.rules_level3.**Asn_Lv3WFSCMB**(**args*, ***kwargs*)

Bases: AsnMixin_Science

Level 3 Wavefront Control & Sensing Association

For coarse and fine phasing, dither pairs need to be associated to be combined. The optical path is assumed to be equivalent within an activity.

Characteristics:

- Association type: wfs-image3
- Pipeline: calwebb_wfs-image3
- Coarse and fine phasing dithers

Attributes Summary

dms_product_name	Define product name

Attributes Documentation

dms_product_name

Define product name

Modification is to append the expspcin value after the calibration suffix.

Asn_Lv3WFSSNIS

class jwst.associations.lib.rules_level3.**Asn_Lv3WFSSNIS**(**args*, ***kwargs*)

Bases: AsnMixin_Spectrum

Level 3 WFSS/Grism Association

Characteristics:

- Association type: spec3
- Pipeline: calwebb_spec3
- Gather all grism exposures

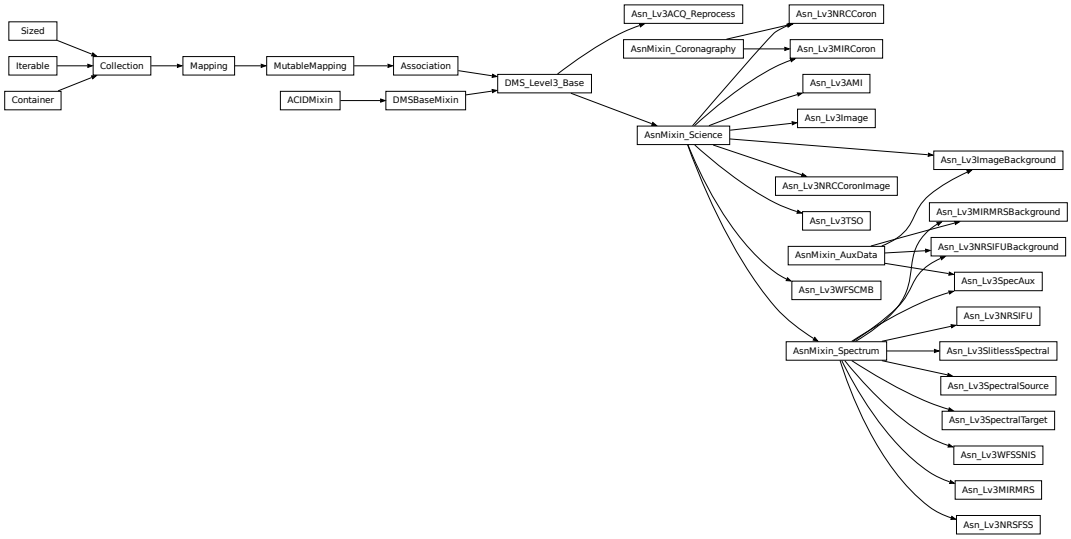
Attributes Summary

dms_product_name	Define product name.

Attributes Documentation

dms_product_name

Class Inheritance Diagram



jwst.associations.lib.dms_base Module

Association attributes common to DMS-based Rules

Classes

Constraint_TargetAcq([association])	Select on target acquisition exposures
Constraint_TSO(*args, **kwargs)	Match on Time-Series Observations
Constraint_WFSC(*args, **kwargs)	Match on Wave Front Sensing and Control Observations
DMSBaseMixin(*args, **kwargs)	Association attributes common to DMS-based Rules

Constraint_TargetAcq

class jwst.associations.lib.dms_base.**Constraint_TargetAcq**(*association=None*)

Bases: *SimpleConstraint*

Select on target acquisition exposures

Parameters

association (Association) – If specified, use the `get_exposure_type` method of the association rather than the utility version.

Constraint_TSO

class jwst.associations.lib.dms_base.**Constraint_TSO**(**args*, ***kwargs*)

Bases: *Constraint*

Match on Time-Series Observations

Constraint_WFSC

class jwst.associations.lib.dms_base.**Constraint_WFSC**(**args*, ***kwargs*)

Bases: *Constraint*

Match on Wave Front Sensing and Control Observations

DMSBaseMixin

class jwst.associations.lib.dms_base.**DMSBaseMixin**(**args*, ***kwargs*)

Bases: `ACIDMixin`

Association attributes common to DMS-based Rules

sequence

The sequence number of the current association

Type

int (https://docs.python.org/3/library/functions.html#int)

Attributes Summary

acid	Association ID
asn_name	The association name
current_product	
from_items	The list of items that contributed to the association.
member_ids	Set of all member ids in all products of this association
validity	Keeper of the validity tests

Methods Summary

create(item[, version_id])	Create association if item belongs
get_exposure_type(item[, default])	Determine the exposure type of a pool item
is_item_ami(item)	Is the given item AMI (NIRISS Aperture Masking Interferometry)
is_item_coron(item)	Is the given item Coronagraphic
is_item_member(item)	Check if item is already a member of this association
is_item_tso(item[, other_exp_types])	Is the given item TSO
is_member(new_member)	Check if member is already a member
item_getattr(item, attributes)	Return value from any of a list of attributes
new_product([product_name])	Start a new product
reset_sequence()	
update_asn([item, member])	Update association meta information
update_degraded_status()	Update association degraded status
update_validity(entry)	
validate(asn)	

Attributes Documentation

acid

Association ID

asn_name

The association name

The name that identifies this association. When dumped, will form the basis for the suggested file name.

Typically, it is generated based on the current state of the association, but can be overridden.

current_product

from_items

The list of items that contributed to the association.

member_ids

Set of all member ids in all products of this association

validity

Keeper of the validity tests

Methods Documentation

classmethod create(*item*, *version_id=None*)

Create association if item belongs

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to initialize the association with.
- **version_id** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – Version_Id to use in the name of this association. If None, nothing is added.

Returns

2-tuple consisting of:

- association : The association or, if the item does not match this rule, None
- [ProcessList[, ...]]: List of items to process again.

Return type
(association, reprocess_list)

get_exposure_type(*item*, *default='science'*)

Determine the exposure type of a pool item

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The pool entry to determine the exposure type of
- **default** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – The default exposure type. If None, routine will raise LookupError

Returns

exposure_type –

Exposure type. Can be one of

- 'science': Item contains science data
- 'target_acquisition': Item contains target acquisition data.
- 'autoflat': NIRSpec AUTOFLAT
- 'autowave': NIRSpec AUTOWAVE
- 'psf': PSF
- 'imprint': MSA/IFU Imprint/Leakcal

Return type
str (https://docs.python.org/3/library/stdtypes.html#str)

Raises
LookupError (https://docs.python.org/3/library/exceptions.html#LookupError) – When `default` is None and an exposure type cannot be determined

is_item_ami(*item*)

Is the given item AMI (NIRISS Aperture Masking Interferometry)

Determine whether the specific item represents AMI data or not. This simply includes items with EXP_TYPE='NIS_AMI'.

Parameters
item (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to check for.

Returns
is_item_ami – Item represents an AMI exposure.

Return type
bool (https://docs.python.org/3/library/functions.html#bool)

is_item_coron(*item*)

Is the given item Coronagraphic

Determine whether the specific item represents true Coronagraphic data or not. This will include all items in CORON_EXP_TYPES (both NIRCam and MIRI), **except** for NIRCam short-wave detectors included in a coronagraphic exposure but do not have an occulter in their field-of-view.

Parameters
item (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to check for.

Returns
is_item_coron – Item represents a true Coron exposure.

Return type
bool (https://docs.python.org/3/library/functions.html#bool)

is_item_member(*item*)

Check if item is already a member of this association

Parameters
item (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to check for.

Returns
is_item_member – True if item is a member.

Return type
bool (https://docs.python.org/3/library/functions.html#bool)

is_item_tso(*item*, *other_exp_types=None*)

Is the given item TSO

Determine whether the specific item represents TSO data or not. This is used to determine the naming of files, i.e. “rate” vs “rateints” and “cal” vs “calints”.

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to check for.
- **other_exp_types** (*[str* (https://docs.python.org/3/library/stdtypes.html#str)*[,...]] or None*) – List of other exposure types to consider TSO-like.

Returns
is_item_tso – Item represents a TSO exposure.

Return type
bool (https://docs.python.org/3/library/functions.html#bool)

is_member(*new_member*)

Check if member is already a member

Parameters
new_member (*Member*) – The member to check for

item_getattr(*item*, *attributes*)

Return value from any of a list of attributes

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – item to retrieve from
- **attributes** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – List of attributes

Returns

Returns the value and the attribute from which the value was taken.

Return type

(attribute, value)

Raises

KeyError (https://docs.python.org/3/library/exceptions.html#KeyError) – None of the attributes are found in the dict.

new_product(*product_name='undefined'*)

Start a new product

classmethod reset_sequence()

update_asn(*item=None*, *member=None*)

Update association meta information

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict) *or None*) – Item to use as a source. If not given, item-specific information will be left unchanged.
- **member** (*Member or None*) – An association member to use as source. If not given, member-specific information will be update from current association/product membership.

Notes

If both `item` and `member` are given, information in `member` will take precedence.

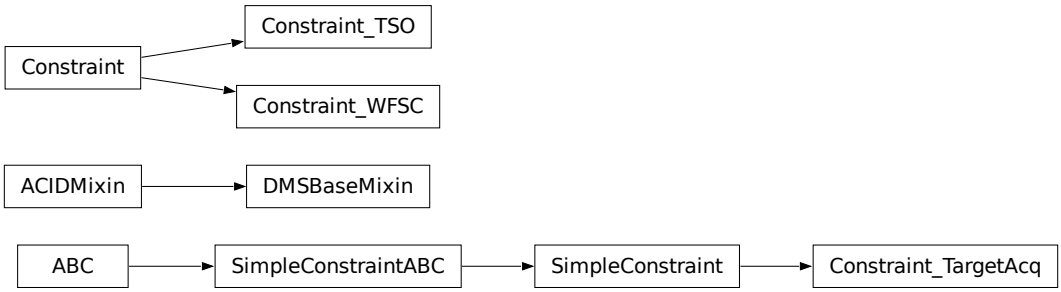
update_degraded_status()

Update association degraded status

update_validity(*entry*)

classmethod validate(*asn*)

Class Inheritance Diagram



jwst.associations.lib.constraint Module

Constraints

Classes

AttrConstraint([init, sources, evaluate, ...])	Test attribute of an item
Constraint([init, reduce, name, ...])	Constraint that is made up of SimpleConstraints
ConstraintTrue([init, value, name])	Always return True
SimpleConstraint([init, sources, ...])	A basic constraint

AttrConstraint

class jwst.associations.lib.constraint.**AttrConstraint**(*init=None*, *sources=None*, *evaluate=False*, *force_reprocess=False*, *force_undefined=False*, *force_unique=True*, *invalid_values=None*, *only_on_match=False*, *onlyif=None*, *required=True*, ***kwargs*)

Bases: SimpleConstraintABC

Test attribute of an item

Parameters

- **sources** (*[str* (https://docs.python.org/3/library/stdtypes.html#str)*[,...]]*) – List of attributes to query
- **value** (*str* (https://docs.python.org/3/library/stdtypes.html#str)*, function or None*) – The value to check for. If None and `force_unique`, any value in the first available source will become the value. If function, the function takes no arguments and returns a string.
- **evaluate** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Evaluate the item's value before checking condition.

- **force_reprocess** (*ListCategory.state or False*) – Add item back onto the reprocess list using the specified *ProcessList* work over state.
- **force_unique** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If the initial value is None (https://docs.python.org/3/library/constants.html#None) or a list of possible values, the constraint will be modified to be the value first matched.
- **invalid_values** (*[str* (https://docs.python.org/3/library/stdtypes.html#str)*[,...]]*) – List of values that are invalid in an item. Will cause a non-match.
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – Name of the constraint.
- **only_on_match** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `force_reprocess`, only do the reprocess if the entire constraint is satisfied.
- **onlyif** (*function*) – Boolean function that takes `item` as argument. If True, the rest of the condition is checked. Otherwise return as a matched condition
- **required** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – One of the sources must exist. Otherwise, return as a matched constraint.

found_values

Set of actual found values for this condition.

Type
set (https://docs.python.org/3/library/stdtypes.html#set)(str (https://docs.python.org/3/library/stdtypes.html#str)[,...])

matched

Last result of *check_and_set*

Type
bool (https://docs.python.org/3/library/functions.html#bool)

Methods Summary

check_and_set(item)	Check and set constraints based on item

Methods Documentation

check_and_set(*item*)

Check and set constraints based on item

Parameters
item (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to check on.

Returns

success, reprocess –

Returns 2-tuple of

- True if check is successful.
- List of *ProcessList*.

Return type
bool (https://docs.python.org/3/library/functions.html#bool), [ProcessList[,...]]

Constraint

class jwst.associations.lib.constraint.**Constraint**(*init=None*, *reduce=None*, *name=None*, *reprocess_on_match=False*, *reprocess_on_fail=False*, *work_over=ListCategory.BOTH*, *reprocess_rules=None*)

Bases: `object` (https://docs.python.org/3/library/functions.html#object)

Constraint that is made up of SimpleConstraints

Parameters

- **init** (*object* (https://docs.python.org/3/library/functions.html#object) *or [object* (https://docs.python.org/3/library/functions.html#object)*[,...]]*) – A single object or list of objects where the objects are as follows. - SimpleConstraint or subclass - Constraint
- **reduce** (*function*) – A reduction function with signature `x(iterable)` where `iterable` is the `components` list. Returns boolean indicating state of the components. Default value is *Constraint.all*
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – Optional name for constraint.
- **reprocess_on_match** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Reprocess the item if the constraint is satisfied.
- **reprocess_on_fail** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Reprocess the item if the constraint is not satisfied.
- **work_over** (*ListCategory.[BOTH, EXISTING, RULES]*) – The condition on which this constraint should operate.
- **reprocess_rules** (*[rule[,..]] or None*) – List of rules to be applied to. If None, calling function will determine the ruleset. If empty, [], all rules will be used.

constraints

List of *Constraint* or *SimpleConstraint* that make this constraint.

Type
[*Constraint*[,...]]

matched

Result of the last *check_and_set*

Type
bool (https://docs.python.org/3/library/functions.html#bool)

reduce

A reduction function with signature `x(iterable)` where `iterable` is the `components` list. Returns boolean indicating state of the components. Predefined functions are: - *all*: True if all components return True - *any*: True if any component returns True

Type
function

Notes

Named constraints can be accessed directly through indexing:

```
>>> c = Constraint(SimpleConstraint(name='simple', value='a_value'))
>>> c['simple']
SimpleConstraint({'sources': <function SimpleConstraint.__init__.<locals>.<lambda>␣
↪at 0x7f8be05f5730>,
                  'force_unique': True,
                  'test': <bound method SimpleConstraint.eq of SimpleConstraint({...
↪})>,
                  'reprocess_on_match': False,
                  'reprocess_on_fail': False,
                  'work_over': 1,
                  'reprocess_rules': None,
                  'value': 'a_value',
                  'name': 'simple',
                  'matched': False})
```

Attributes Summary

dup_names	Return dictionary of constraints with duplicate names
id	Return identifyer for the constraint

Methods Summary

all(item, constraints)	Return positive only if all results are positive.
any(item, constraints)	Return the first successful constraint.
append(constraint)	Append a new constraint
check_and_set(item[, work_over])	Check and set the constraint
copy()	Copy ourselves
get_all_attr(attribute)	Return the specified attribute
notall(item, constraints)	True if not all of the constraints match
notany(item, constraints)	True if none of the constraints match

Attributes Documentation

dup_names

Return dictionary of constraints with duplicate names

This method is meant to be overridden by classes that need to traverse a list of constraints.

Returns

dups – Returns a mapping between the duplicated name and all the constraints that define that name.

Return type

{str: [constraint[,...]][,...]}

id

Return identifyer for the constraint

Returns
id – The identifyer

Return type
str (https://docs.python.org/3/library/stdtypes.html#str)

Methods Documentation

static all(*item*, *constraints*)

Return positive only if all results are positive.

static any(*item*, *constraints*)

Return the first successful constraint.

append(*constraint*)

Append a new constraint

check_and_set(*item*, *work_over=ListCategory.BOTH*)

Check and set the constraint

Returns

success, reprocess –

Returns 2-tuple of

- success : True if check is successful.
- List of *ProcessList*.

Return type
bool (https://docs.python.org/3/library/functions.html#bool), [ProcessList[,...]]

copy()

Copy ourselves

get_all_attr(*attribute: str (https://docs.python.org/3/library/stdtypes.html#str)*)

Return the specified attribute

This method is meant to be overridden by classes that need to traverse a list of constraints.

Parameters
attribute (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The attribute to retrieve

Returns
result – The list of values of the attribute in a tuple. If there is no attribute, an empty tuple is returned.

Return type
[(*SimpleConstraint* or *Constraint*, object (https://docs.python.org/3/library/functions.html#object))[,...]]

Raises
AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) – If the attribute is not found.

static notall(*item*, *constraints*)

True if not all of the constraints match

static notany(*item*, *constraints*)

True if none of the constraints match

ConstraintTrue

class jwst.associations.lib.constraint.**ConstraintTrue**(*init=None*, *value=None*, *name=None*, ***kwargs*)

Bases: SimpleConstraintABC

Always return True

Methods Summary

check_and_set(item)	Check and set the constraint

Methods Documentation

check_and_set(*item*)

Check and set the constraint

Returns

success, reprocess –

Returns 2-tuple of

- True if check is successful.
- List of *ProcessList*.

Return type

bool (https://docs.python.org/3/library/functions.html#bool), [ProcessList[,...]]

SimpleConstraint

class jwst.associations.lib.constraint.**SimpleConstraint**(*init=None*, *sources=None*, *force_unique=True*, *test=None*, *reprocess_on_match=False*, *reprocess_on_fail=False*, *work_over=ListCategory.BOTH*, *reprocess_rules=None*, ***kwargs*)

Bases: SimpleConstraintABC

A basic constraint

Parameters

- **init** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – dict where the key:value pairs define the following parameters
- **value** (*object* (https://docs.python.org/3/library/functions.html#object) *or None*) – Value that must be matched. If None, any retrieved value will match.

- **sources** (*func(item) or None*) – Function taking `item` as argument used to retrieve a value to check against. If None, the item itself is used as the value.
- **force_unique** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If the constraint is satisfied, reset `value` to the value of the source.
- **test** (*function*) – The test function for the constraint. Takes two arguments:
  - constraint
  - object to compare against.

  Returns a boolean. Default is *SimpleConstraint.eq*
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – Option name for constraint
- **reprocess_on_match** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Reprocess the item if the constraint is satisfied.
- **reprocess_on_fail** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Reprocess the item if the constraint is not satisfied.
- **work_over** (*ListCategory.[BOTH, EXISTING, RULES]*) – The condition on which this constraint should operate.
- **reprocess_rules** (*[rule[,..]] or None*) – List of rules to be applied to. If None, calling function will determine the ruleset. If empty, [], all rules will be used.

All `Parameters` are also `Attributes`

Examples

Create a constraint where the attribute `attr` of an object matches the value `my_value`:

```
>>> c = SimpleConstraint(value='my_value')
>>> print(c)
SimpleConstraint({'name': None, 'value': 'my_value'})
```

To check a constraint, call *check_and_set*. A successful match will return a tuple of `True` (https://docs.python.org/3/library/constants.html#True) and a reprocess list. >>> item = 'my_value' >>> c.check_and_set(item) (True, [])

If it doesn't match, `False` (https://docs.python.org/3/library/constants.html#False) will be returned. >>> bad_item = 'not_my_value' >>> c.check_and_set(bad_item) (False, [])

A *SimpleConstraint* can also be initialized by a `dict` (https://docs.python.org/3/library/stdtypes.html#dict) of the relevant parameters: >>> init = {'value': 'my_value'} >>> c = SimpleConstraint(init) >>> print(c) SimpleConstraint({'name': None, 'value': 'my_value'})

If the value to check is `None` (https://docs.python.org/3/library/constants.html#None), the *SimpleConstraint* will successfully match whatever object given. However, a new *SimpleConstraint* will be returned where the `value` is now set to whatever the attribute was of the object. >>> c = SimpleConstraint(value=None) >>> matched, reprocess = c.check_and_set(item) >>> print(c) SimpleConstraint({'name': None, 'value': 'my_value'})

This behavior can be overridden by the `force_unique` parameter: >>> c = SimpleConstraint(value=None, force_unique=False) >>> matched, reprocess = c.check_and_set(item) >>> print(c) SimpleConstraint({'name': None, 'value': None})

Methods Summary

check_and_set(item)	Check and set the constraint
eq(value1, value2)	True if constraint.value and item are equal.

Methods Documentation

check_and_set(*item*)

Check and set the constraint

Returns

success, reprocess –

Returns 2-tuple of

- True if check is successful.
- List of *ProcessList*.

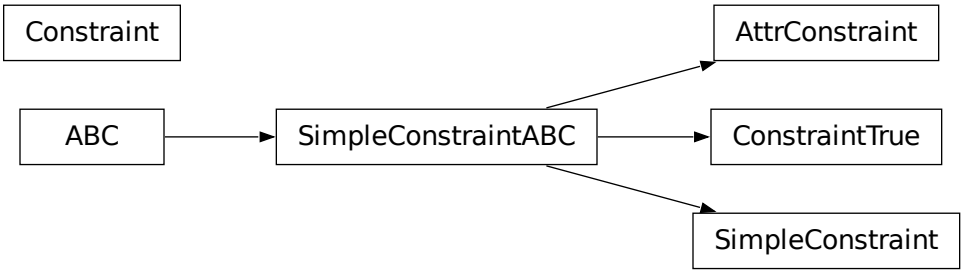
Return type

bool (https://docs.python.org/3/library/functions.html#bool), [ProcessList[,...]]

eq(*value1*, *value2*)

True if constraint.value and item are equal.

Class Inheritance Diagram



15.1.8 Background Step

Description

Class
jwst.background.BackgroundStep

Alias
background

The background subtraction step performs image-from-image subtraction in order to accomplish subtraction of background signal. The step takes as input one target exposure, to which the subtraction will be applied, and a list of one or more background exposures. Two different approaches to background image subtraction are used, depending on the observing mode. Imaging and most spectroscopic modes use one method, while a special method is used for Wide-Field Slitless Spectroscopy (WFSS).

This type of background subtraction is just one method available within the JWST pipeline. See *Background Subtraction* for an overview of all the methods and to which observing modes they're applicable.

Imaging and Non-WFSS Spectroscopic Modes

If more than one background exposure is provided, they will be averaged together before being subtracted from the target exposure. Iterative sigma clipping is applied during the averaging process, to reject sources or other outliers. The clipping is accomplished using the function astropy.stats.sigma_clip (http://docs.astropy.org/en/stable/api/astropy.stats.sigma_clip.html). The background step allows users to supply values for the `sigma_clip` parameters `sigma` and `maxiters` (see *Step Arguments*), in order to control the clipping operation.

For imaging mode observations, the calculation of the average background image depends on whether the background exposures are "rate" (2D) or "rateint" (3D) exposures. In the case of "rate" exposures, the average background image is produced as follows:

1. Clip the combined SCI arrays of all background exposures. For mixtures of full chip and subarray data, only overlapping regions are used
2. Compute the mean of the unclipped SCI values
3. Sum in quadrature the ERR arrays of all background exposures, clipping the same input values as determined for the SCI arrays, and convert the result to an uncertainty in the mean
4. Combine the DQ arrays of all background exposures using a bitwise OR operation

In the case of "rateint" exposures, each background exposure can have multiple integrations, so calculations are slightly more involved. The "overall" average background image is produced as follows:

1. Clip the SCI arrays of each background exposure along its integrations
2. Compute the mean of the unclipped SCI values to yield an average image for each background exposure
3. Clip the means of all background exposure averages
4. Compute the mean of the unclipped background exposure averages to yield the "overall" average background image
5. Sum in quadrature the ERR arrays of all background exposures, clipping the same input values as determined for the SCI arrays, and convert the result to an uncertainty in the mean (This is not yet implemented)
6. Combine the DQ arrays of all background exposures, by first using a bitwise OR operation over all integrations in each exposure, followed by doing by a bitwise OR operation over all exposures.

The average background exposure is then subtracted from the target exposure. The subtraction consists of the following operations:

1. The SCI array of the average background is subtracted from the SCI array of the target exposure
2. The ERR array of the target exposure is currently unchanged, until full error propagation is implemented in the entire pipeline
3. The DQ arrays of the average background and the target exposure are combined using a bitwise OR operation

If the target exposure is a simple ImageModel, the background image is subtracted from it. If the target exposure is in the form of a 3-D CubeModel (e.g. the result of a time series exposure), the average background image is subtracted from each plane of the CubeModel.

The combined, averaged background image can be saved using the step parameter `save_combined_background`.

WFSS Mode

For Wide-Field Slitless Spectroscopy expsoures (NIS_WFSS and NRC_WFSS), a background reference image is subtracted from the target exposure. Before being subtracted, the background reference image is scaled to match the signal level of the WFSS image within background (source-free) regions of the image.

The locations of source spectra are determined from a source catalog (specified by the primary header keyword SCATFILE), in conjunction with a reference file that gives the wavelength range (based on filter and grism) that is relevant to the WFSS image. All regions of the image that are free of source spectra are used for scaling the background reference image. The step argument `wfss_mmag_extract` can be used, if desired, to set the minimum (faintest) abmag of the source catalog objects used to define the background regions. The default is to use all source catalog entries that result in a spectrum falling within the WFSS image.

Robust mean values are obtained for the background regions in the WFSS image and for the same regions in the background reference image, and the ratio of those two mean values is used to scale the background reference image. The robust mean is computed by excluding the lowest 25% and highest 25% of the data (using the numpy.percentile function), and taking a simple arithmetic mean of the remaining values. Note that NaN values (if any) in the background reference image are currently set to zero. If there are a lot of NaNs, it may be that more than 25% of the lowest values will need to be excluded.

For both background methods the output results are always returned in a new data model, leaving the original input model unchanged.

Upon successful completion of the step, the S_BKDSUB keyword will be set to "COMPLETE" in the output product.

Step Arguments

The background image subtraction step has four optional arguments. The first two are used only when the step is applied to non-WFSS exposures. They are used in the process of creating an average background image, to control the sigma clipping, and are passed as arguments to the astropy `sigma_clip` function:

`--sigma`
: When combining multiple background images, the number of standard deviations to use for the clipping limit. Defaults to 3.

`--maxiters`
: When combining multiple background images, the number of clipping iterations to perform, or `None` to clip until convergence is achieved. Defaults to `None`.

`--save_combined_background`
: Saves the combined background image used for background subtraction. Defaults to `False`.

`--wfss_mmag_extract`
: Only applies to Wide Field Slitless Spectroscopy (WFSS) exposures. Sets the minimum (faintest) magnitude

limit to use when selecting sources from the WFSS source catalog, based on the value of `isophotal_abmag` in the source catalog. Defaults to `None`.

Reference Files

The background image subtraction step uses reference files only when processing Wide-Field Slitless Spectroscopy (WFSS) exposures. Two reference files are used for WFSS mode: WFSSBKG and WAVELENGTHRANGE. The WAVELENGTHRANGE reference file is used in the process of determining the locations of source spectra in the image, and conversely the image areas that contain only background signal.

WFSS Background reference file

REFTYPE
WFSSBKG

Data model
`WfssBkgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.WfssBkgModel.html#jwst.datamodels.W

The WFSS background reference file contains a "master" image of the dispersed background produced by a particular filter+grism combination.

Reference Selection Keywords for WFSSBKG

CRDS selects appropriate WFSSBKG references based on the following keywords. WFSSBKG is not applicable for instruments not in the table.

Instrument	Keywords
NIRCam	INSTRUME, EXP_TYPE, DETECTOR, FILTER, PUPIL, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DETECTOR, FILTER, PUPIL, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for WFSSBKG

In addition to the standard reference file keywords listed above, the following keywords are *required* in WFSSBKG reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for WFSSBKG*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector
EXP_TYPE	model.meta.exposure.type
FILTER	model.meta.instrument.filter
PUPIL	model.meta.instrument.pupil

Reference File Format

WFSSBKG reference files are FITS files with 3 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The characteristics of the FITS extensions are as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float
ERR	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	integer
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

WAVELENGTHRANGE Reference File

REFTYPE
WAVELENGTHRANGE

Data model
`WavelengthrangeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.WavelengthrangeModel.html#j

The WAVELENGTHRANGE reference file contains information on the minimum and maximum wavelengths of various spectroscopic modes, which can be order-dependent. The reference data are used to construct bounding boxes around the spectral traces produced by each object in the NIRCam and NIRISS WFSS modes. If a list of `GrismObject` is supplied, then no reference file is necessary.

Reference Selection Keywords for WAVELENGTHRANGE

CRDS selects appropriate WAVELENGTHRANGE references based on the following keywords. WAVELENGTHRANGE is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for WAVELENGTHRANGE

In addition to the standard reference file keywords listed above, the following keywords are *required* in WAVELENGTHRANGE reference files

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

These keywords are used as CRDS selectors

Reference Selection Keywords for WAVELENGTHRANGE

CRDS selects appropriate WAVELENGTHRANGE references based on the following keywords. WAVELENGTHRANGE is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Reference File Format

WAVELENGTHRANGE reference files are in ASDF format, with the format and contents specified by the `WavelengthrangeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.WavelengthrangeModel.html#jwst.datamoc data model schema. The exact content varies a bit depending on instrument mode, as explained in the following sections.

MIRI MRS

For MIRI MRS, the WAVELENGTHRANGE file consists of two fields that define the wavelength range for each combination of channel and band.

channels
: An ordered list of all possible channel and band combinations for MIRI MRS, e.g. "1SHORT".

wavelengthrange
: An ordered list of (lambda_min, lambda_max) for each item in the list above

NIRSpec

For NIRSpec, the WAVELENGTHRANGE file is a dictionary storing information about default wavelength range and spectral order for each combination of filter and grating.

filter_grating

order
: Default spectral order

range
: Default wavelength range

NIRCam WFSS, NIRCam TSGRISM, NIRISS WFSS

For NIRCam WFSS and TSGRIM modes, as well as NIRISS WFSS mode, the WAVELENGTHRANGE reference file contains the wavelength limits to use when calculating the minimum and maximum dispersion extents on the detector. It also contains the default list of orders that should be extracted for each filter. To be consistent with other modes, and for convenience, it also lists the orders and filters that are valid with the file.

order
A list of orders this file covers

wavelengthrange
A list containing the list of [order, filter, wavelength min, wavelength max]

waverange_selector
The list of FILTER names available

extract_orders
A list containing the list of orders to extract for each filter

jwst.background Package

Classes

BackgroundStep([name, parent, config_file, ...])	BackgroundStep: Subtract background exposures from target exposures.

BackgroundStep

class jwst.background.**BackgroundStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

BackgroundStep: Subtract background exposures from target exposures.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

bkg_suffix
class_alias
reference_file_types
spec

Methods Summary

process(input, bkg_list)	Subtract the background signal from target exposures by subtracting designated background images from them.

Attributes Documentation

bkg_suffix = 'combinedbackground'

class_alias = 'background'

reference_file_types = ['wfssbkg', 'wavelengthrange']

spec

```
save_combined_background = boolean(default=False)  # Save combined background␣
↪image
sigma = float(default=3.0)  # Clipping threshold
maxiters = integer(default=None)  # Number of clipping iterations
wfss_mmag_extract = float(default=None)  # WFSS minimum abmag to extract
```

Methods Documentation

process(*input*, *bkg_list*)

Subtract the background signal from target exposures by subtracting designated background images from them.

Parameters

- **input** (*JWST data model*) – input target data model to which background subtraction is applied
- **bkg_list** (*filename list*) – list of background exposure file names

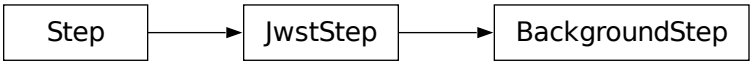
Returns

result – the background-subtracted target data model

Return type

JWST data model

Class Inheritance Diagram



15.1.9 Background Subtraction

Introduction

Subtraction of background signal can take several different forms depending on the observing mode and the available data. Here we give an overview of the different methods that are available, when they can be used, and where they occur in the processing flow. Imaging and spectroscopic observations share one method for background subtraction, while others are unique to spectroscopic data only. See the documentation for the individual steps mentioned here for complete details on how each of them function.

Imaging Mode

Background subtraction for imaging data is currently available in several places within the calibration pipeline stages.

1. Image-from-image subtraction can be performed by the *background* step during *calwebb_image2* processing. The background images come from observations of a dedicated background target.
2. Background matching and subtraction can be performed within an ensemble of images by the *skymatch* step during *calwebb_image3* processing.
3. Local background subtraction for individual sources can be performed by the *source_catalog* step within the *calwebb_image3* pipeline.

Spectroscopic Modes

Spectroscopic observations allow for some additional ways of performing background subtraction. The list of options includes:

1. Image-from-image subtraction can be performed by the *background* step during *calwebb_spec2* processing. The background images can come from:
   a) Observations of a dedicated background target
   b) Nodded observations of a point-like science target
2. Subtraction of a "master" background spectrum, where the master background spectrum can come from:
   a) Observations of a dedicated background target
   b) Nodded observations of a point-like science target
   c) Dedicated background slitlets in a NIRSpec MOS exposure
   d) A user-supplied spectrum

3. Local background subtraction for individual spectral can be performed by the *extract_1d* step when doing 1D spectral extraction.

The following table shows the list of image-from-image and master background subtraction methods available for various spectroscopic observation modes, and indicates the pipeline and step in which the subtraction operation occurs. The table also shows which method is applied by default in the operational pipeline when the available data support multiple methods.

Note: Master background subtraction is applied in the *calwebb_spec3* pipeline for most spectroscopic modes, but for **NIRSpec MOS** mode it is applied during *calwebb_spec2* processing.

Mode	calwebb_spec2 background	calwebb_spec3 master_background	calwebb_spec2 master_background_nrs_slits
NIRSpec Fixed Slit:			
Dedicated background	Default	Optional	
Nodded point source	Default	Optional	
User supplied		Default	
NIRSpec IFU:			
Dedicated background	Default	Optional	
Nodded point source	Default	Optional	
User supplied		Default	
NIRSpec MOS:			
Background slitlets			Default
Nodded point source	Default		
User supplied			Default
MIRI LRS Fixed Slit:			
Dedicated background	Default	Optional	
Nodded point source	Default	Optional	
User supplied		Default	
MIRI MRS:			
Dedicated background	Default	Optional	
Nodded point source	Default	Optional	
User supplied		Default	

These background subtraction methods are only available for the observing modes listed in the table. Other spectroscopic modes, including NIRCam and NIRISS Wide Field Slitless Spectroscopy (WFSS), NIRCam Time Series Grism, NIRISS Single Object Slitless Spectroscopy (SOSS), and MIRI LRS slitless, use other ways of handling background.

Image-from-Image Subtraction

As explained in the documentation for the *background* step, this process combines one or more exposures to be treated as backgrounds into a sigma-clipped mean background image, which is then directly subtracted, in detector space, from an exposure being processed in the *calwebb_image2* or *calwebb_spec2* pipelines for imaging or spectroscopic data, respectively. For imaging mode observations this is only possible when observations of a designated background target have been obtained. For spectroscopic modes this is possible either through observations of a designated background target or when nodded exposures of a point-like target are obtained (e.g. using the MIRI LRS "ALONG-SLIT-NOD" dither pattern for an LRS fixed slit observation). Exposures from one nod position can be used as background for exposures at the other nod position, assuming the source is point-like.

In either instance, the exposures to be used as background are included in the `image2` or `spec2` ASN file used to process the science target exposures, where the background exposures are labeled with an ASN member type of "background".

Spectroscopic observations that have designated background target exposures or nodded exposures can use either the image-from-image or master background subtraction methods. In the operational pipeline the image-from-image subtraction method is applied by default and the master background subtraction is skipped. A user has the option to reprocess the data and apply the other method, if desired.

Master Background Subtraction

In general, the master background subtraction method works by taking a 1D background spectrum, interpolating it back into the 2D space of a science image, and then subtracting it. This allows for higher SNR background data to be used than what might be obtainable by doing direct image-from-image subtraction using only one or a few background images. The 1D master background spectrum can either be constructed on-the-fly by the calibration pipeline from available background data or supplied by the user. See the documentation for the *master background subtraction* step for full details.

As with image-from-image subtraction, there are different ways of obtaining the data necessary for constructing a master background spectrum, depending on the observing mode:

1. Observations of a designated background target
2. Nodded observations of a point-like source
3. Dedicated background slitlets in a NIRSpec MOS exposure
4. User-supplied master background spectrum

All of these scenarios apply the master background subtraction during *calwebb_spec3* processing, except for NIRSpec MOS observations. Master background subtraction for NIRSpec MOS, using either data from background slitlets contained in each MOS exposure or a user-supplied master background spectrum, is applied during *calwebb_spec2*, due to unique methods that must be used for MOS exposures.

For scenarios that apply master background subtraction during *calwebb_spec3* processing, the fully-calibrated 1D spectra ("x1d" products) from either dedicated background target exposures or nodded science exposures are used by the *master_background* step to construct the 1D master background spectrum. These are the x1d products created during the last step of the preceding *calwebb_spec2* pipeline when it is used to process each exposure. Again, see the documentation for the *master background subtraction* step for full details of the source of the background data for these scenarios.

If the user supplies a 1D master background spectrum, the construction of the master background spectrum in the pipeline is skipped and the user-supplied spectrum is used in its place. This applies to all modes, including NIRSpec MOS.

As mentioned above, NIRSpec MOS observations require special handling to correctly apply master background subtraction. If a MOS observation uses an MSA configuration that includes one or more slitlets containing only background signal, the background slitlets are fully calibrated and extracted to produce one or more 1D background spectra. The background spectra are combined into a 1D master background spectrum, which is then interpolated back into the 2D

space of all slitlets and subtracted. If the user supplies a master background spectrum for a MOS observation, that spectrum is used to do the subtraction. Again note that for NIRSpec MOS mode these operations take place during *calwebb_spec2* pipeline processing, not *calwebb_spec3* like all other modes.

15.1.10 Barshadow Correction

Description

Class
jwst.barshadow.BarShadowStep

Alias
barshadow

Overview

The `barshadow` step calculates the correction to be applied to NIRSpec MSA data for extended sources due to the bar that separates adjacent microshutters. This correction is applied to MultiSlit data after the *pathloss* correction has been applied in the *calwebb_spec2* pipeline.

Input details

The input data must have been processed through the *extract_2d* step, so that cutouts have been created for each of the slitlets used in the exposure. Hence the input must be in the form of a `MultiSlitModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamodels.MultiSlitModel).

It is also assumed that the input data have been processed through the *srctype* step, which for NIRSpec MSA exposures sets the SRCTYPE keyword value for each slit to "POINT", "EXTENDED", or "UNKNOWN." If the source type is "EXTENDED" or "UNKNOWN", or the SRCTYPE keyword is not present, the default action is to treat the source as extended and apply the `barshadow` correction. If SRCTYPE="POINT" for a given slit, the correction is not applied.

Algorithm

The step loops over all slit instances contained in the input exposure, computing and applying the barshadow correction to each slit for which the source type has been determined to be extended.

The *BARSHADOW Reference File* contains the correction as a function of Y and wavelength for a single open shutter (the DATA1X1 extension), and for 2 adjacent open shutters (DATA1X3). This allows on-the-fly construction of a model for any combination of open and closed shutters. The shutter configuration of a slitlet is contained in the attribute shutter_state, which shows whether the shutters of the slitlet are open, closed, or contain the source. Once the correction as a function of Y and wavelength is calculated, the WCS transformation from the detector to the slit frame is used to calculate Y and wavelength for each pixel in the cutout. The Y values are scaled from shutter heights to shutter spacings, and then the Y and wavelength values are interpolated into the model to determine the correction for each pixel.

Once the 2-D correction array for a slit has been computed, it is applied to the science (SCI), error (ERR), and variance (VAR_POISSON, VAR_RNOISE, and VAR_FLAT) data arrays of the slit. The correction values are divided into the SCI and ERR arrays, and the square of the correction values are divided into the variance arrays.

Output product

The output is a new copy of the input `MultiSlitModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitMode with the corrections applied to the slit data arrays. The 2-D correction array for each slit is also added to the datamodel in the "BARSHADOW" extension.

Step Arguments

The `barshadow` step has the following optional arguments.

`--inverse` (boolean, default=False)
A flag to indicate whether the math operations used to apply the correction should be inverted (i.e. multiply the correction into the science data, instead of the usual division).

`--source_type` (string, default=None)
Force the processing to use the given source type (POINT, EXTENDED), instead of using the information contained in the input data.

Reference Files

The `barshadow` step uses a BARSHADOW reference file.

BARSHADOW Reference File

REFTYPE
BARSHADOW

Data model
`BarshadowModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.BarshadowModel.html#jwst.datamodel

Reference Selection Keywords for BARSHADOW

CRDS selects appropriate BARSHADOW references based on the following keywords. BARSHADOW is not applicable for instruments not in the table.

Instrument	Keywords
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for BARSHADOW

In addition to the standard reference file keywords listed above, the following keywords are *required* in BARSHADOW reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for BARSHADOW*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

Reference File Format

BARSHADOW reference files are FITS format, with 4 IMAGE extensions. The FITS primary data array is assumed to be empty. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
DATA1X1	IMAGE	2	101x1001	float
VAR1X1	IMAGE	2	101x1001	float
DATA1X3	IMAGE	2	101x1001	float
VAR1X3	IMAGE	2	101x1001	float

The slow direction has 1001 rows and gives the dependence of the bar shadow correction on the Y location of a pixel from the center of the shutter. The fast direction has 101 rows and gives the dependence of the bar shadow correction of wavelength. The WCS keywords CRPIX1/2, CRVAL1/2 and CDELT1/2 tell how to convert the pixel location in the reference file into a Y location and wavelength. The initial version of the reference file has Y varying from -1.0 for row 1 to +1.0 at row 1001, and the wavelength varying from 0.6×10^{-6} m to 5.3×10^{-6} m.

An example extension header is as follows:

XTENSION	=	'IMAGE '	/	Image extension
BITPIX	=	-64	/	array data type
NAXIS	=	2	/	number of array dimensions
NAXIS1	=	101		
NAXIS2	=	1001		
PCOUNT	=	0	/	number of parameters
GCOUNT	=	1	/	number of groups
EXTNAME	=	'DATA1x1 '	/	extension name
BSCALE	=	1.0		
BZERO	=	0.0		
BUNIT	=	'UNITLESS'		
CTYPE1	=	'METER '		
CTYPE2	=	'UNITLESS'		
CDELT1	=	4.7E-08		
CDELT2	=	0.002		
CRPIX1	=	1.0		
CRPIX2	=	1.0		
CRVAL1	=	6E-07		
CRVAL2	=	-1.0		
APERTURE	=	'MOS1x1 '		
HEIGHT	=	0.00020161		

jwst.barshadow Package

Classes

BarShadowStep([name, parent, config_file, ...])	BarShadowStep: Inserts the bar shadow and wavelength arrays into the data.

BarShadowStep

class jwst.barshadow.**BarShadowStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

BarShadowStep: Inserts the bar shadow and wavelength arrays into the data.

Bar shadow correction depends on the position of a pixel along the slit and the wavelength. It is only applied to uniform sources and only for NRS MSA data.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.

- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	Perform the barshadow correction step

Attributes Documentation

class_alias = 'barshadow'

reference_file_types = ['barshadow']

spec

```
inverse = boolean(default=False)    # Invert the operation
source_type = string(default=None)  # Process as specified source type.
```

Methods Documentation

process(*input*)

Perform the barshadow correction step

Parameters
input (*JWST datamodel*) – input JWST datamodel object

Returns
result – JWST datamodel object with barshadow extension(s) added

Return type
jwst datamodel

Class Inheritance Diagram

15.1.11 Charge Migration

Description

Class
jwst.charge_migration.ChargeMigrationStep

Alias
charge_migration

Overview

This step corrects for an artifact seen in undersampled NIRISS images that may depress flux in resampled images. The artifact is seen in dithered images where the star is centered in a pixel. When the peak pixels of such stars approach the saturation level, they suffer from significant *charge migration*: the spilling of charge from a saturated pixel into its neighboring pixels. This charge migration causes group-to-group differences to decrease significantly once the signal level is greater than ~25,000 ADU. As a result, the last several groups of these ramps get flagged by the *jump* step. The smaller number of groups used for these pixels in the *ramp_fitting* step results in them having larger read noise variances, which in turn leads to lower weights used during resampling. This ultimately leads to a lower than normal flux for the star in resampled images.

Once a group in a ramp has been flagged as affected by charge migration, all subsequent groups in the ramp are also flagged. By flagging these groups, they are not used in the computation of slopes in the *ramp_fitting* step. However, as described in the algorithm section below, they _are_ used in the calculation of the variance of the slope due to readnoise.

Input details

The input must be in the form of a `RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.d

Algorithm

The first group, and all subsequent groups, exceeding the value of the `signal_threshold` parameter is flagged as CHARGELOSS. `signal_threshold` is in units of ADUs. These groups will also be flagged as DO_NOT_USE, and will not be included in the slope calculation during the `ramp_fitting` step. Despite being flagged as DO_NOT_USE, these CHARGELOSS groups are still included in the calculation of the variance due to readnoise. This results in a readnoise variance for undersampled pixels that is similar to that of pixels unaffected by charge migration. For the Poisson noise variance calculation in *ramp_fitting*, the CHARGELOSS/DO_NOT_USE groups are not included.

For integrations having only 1 or 2 groups, no flagging will be performed.

Output product

The output is a new copy of the input RampModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jw
with the updated DQ flags added to the GROUPDQ array.

Arguments

The `charge migration` step has one optional argument that can be set by the user:

- `--signal_threshold`: A floating-point value in units of ADU for the science value above which a group's DQ will be flagged as CHARGELOSS and DO_NOT_USE.

Reference Files

This step does not use any reference files.

jwst.charge_migration Package

Classes

ChargeMigrationStep([name, parent, ...])	This Step sets the CHARGELOSS flag for groups exhibiting significant charge migration.

ChargeMigrationStep

class jwst.charge_migration.**ChargeMigrationStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

This Step sets the CHARGELOSS flag for groups exhibiting significant charge migration.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'charge_migration'

spec

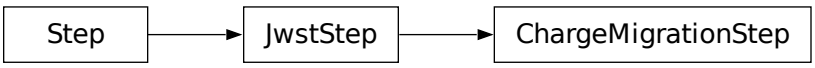
```
signal_threshold = float(default=25000)
skip = boolean(default=True)
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.12 Combine 1D Spectra

Description

Class
jwst.combine_1d.Combine1dStep

Alias
combine_1d

The `combine_1d` step computes a weighted average of 1-D spectra and writes the combined 1-D spectrum as output.

The combination of spectra proceeds as follows. For each pixel of each input spectrum, the corresponding pixel in the output is identified (based on wavelength), and the input value multiplied by the weight is added to the output buffer. Pixels that are flagged (via the DQ column) with "DO_NOT_USE" will not contribute to the output. After all input spectra have been included, the output is normalized by dividing by the sum of the weights.

The weight will typically be the integration time or the exposure time, but uniform (unit) weighting can be specified instead.

The only part of this step that is not completely straightforward is the determination of wavelengths for the output spectrum. The output wavelengths will be increasing, regardless of the order of the input wavelengths. In the ideal case, all input spectra would have wavelength arrays that were very nearly the same. In this case, each output wavelength would be computed as the average of the wavelengths at the same pixel in all the input files. The combine_1d step is intended to handle a more general case where the input wavelength arrays may be offset with respect to each other, or they might not align well due to different distortions. All the input wavelength arrays will be concatenated and then sorted. The code then looks for "clumps" in wavelength, based on the standard deviation of a slice of the concatenated and sorted array of input wavelengths; a small standard deviation implies a clump. In regions of the spectrum where the input wavelengths overlap with somewhat random offsets and don't form any clumps, the output wavelengths are computed as averages of the concatenated, sorted input wavelengths taken N at a time, where N is the number of overlapping input spectra at that point.

Input

An association file specifies which file or files to read for the input data. Each input data file contains one or more 1-D spectra in table format, e.g. as written by the extract_1d step. Each input data file will ordinarily be in MultiSpecModel format (which can contain more than one spectrum).

The association file should have an object called "products", which is a one-element list containing a dictionary. This dictionary contains two entries (at least), one with key "name" and one with key "members". The value for key "name" is a string, the name that will be used as a basis for creating the output file name. "members" is a list of dictionaries, each of which contains one input file name, identified by key "expname".

Output

The output will be in CombinedSpecModel format, with a table extension having the name COMBINE1D. This extension will have eight columns, giving the wavelength, flux, error estimate for the flux, surface brightness, error estimate for the surface brightness, the combined data quality flags, the sum of the weights that were used when combining the input spectra, and the number of input spectra that contributed to each output pixel.

Step Arguments

The `combine_1d` step has one step-specific argument:

--exptime_key
This is a case-insensitive string that identifies the metadata element (or FITS keyword) for the weight to apply to the input data. The default is "integration_time". If the string is "effinttm" or starts with "integration", the integration time (FITS keyword EFFINTTM) is used as the weight. If the string is "effexptm" or starts with "exposure", the exposure time (FITS keyword EFFEXPTM) is used as the weight. If the string is "unit_weight" or "unit weight", the same weight (1) will be used for all input spectra. If the string is anything else, a warning will be logged and unit weight will be used.

Reference File

This step does not use any reference file.

jwst.combine_1d Package

Classes

Combine1dStep([name, parent, config_file, ...])	Combine1dStep: Combine 1-D spectra

Combine1dStep

class jwst.combine_1d.**Combine1dStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Combine1dStep: Combine 1-D spectra

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input_file)	This is where real work happens.

Attributes Documentation

class_alias = 'combine_1d'

spec

```
exptime_key = string(default="exposure_time") # use for weight
```

Methods Documentation

process(*input_file*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.13 Cube Building

Description

Class
`jwst.cube_build.CubeBuildStep`

Alias
cube_build

The `cube_build` step takes MIRI or NIRSpec IFU calibrated 2-D images and produces 3-D spectral cubes. The 2-D disjointed IFU slice spectra are corrected for distortion and assembled into a rectangular cube with three orthogonal axes: two spatial and one spectral.

The `cube_build` step can accept several different forms of input data, including:

1. A single file containing a 2-D IFU image
2. A data model (`IFUImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.data containing a 2-D IFU image
3. An association table (in json format) containing a list of input files
4. A model container with several 2-D IFU data models

There are a number of arguments the user can provide either in a parameter file or on the command line that control the sampling size of the cube, as well as the type of data that is combined to create the cube. See the *Step Arguments* section for more details.

Assumptions

It is assumed that the *assign_wcs* step has been applied to the data, attaching the distortion and pointing information to the image(s). It is also assumed that the *photom* step has been applied to convert the pixel values from units of count rate to surface brightness. This step will only work with MIRI or NIRSpec IFU data. The cube_build algorithm is a flux conserving method, requires the input data to be in units of surface brightness (MJy/sr), and produces 3-D cubes also in units of surface brightness. 1-D spectral extraction from these cubes may then produce spectra either in surface brightness units of MJy/sr or in flux units of Jy.

The NIRSpec calibration plan for point source data is designed to produce units of flux density from the calwebb_spec2 pipeline. For NIRSpec IFU point source data the calwebb_spec2 pipeline divides the flux values by a pixel area map to produce pseudo surface brightness units (MJy/steradian). This allows the cube_build program to conserve flux when it combines and resamples the data. True fluxes are produced only at the *extract_1d_step*, in which a 1D spectrum is extracted from the cube using an appropriate extraction aperture, with resulting units of Jy.

Instrument Information

The JWST integral field unit (IFU) spectrographs obtain simultaneous spectral and spatial data on a relatively compact region of the sky.

The MIRI Medium Resolution Spectrometer (MRS) consists of four IFUs providing simultaneous and overlapping fields of view ranging from ~3.3" x 3.7" to ~7.2" x 7.7" and covering a wavelength range of 5-28 microns. The optics system for the four IFUs is split into two paths. One path is dedicated to the two short wavelength IFUs and the other one handles the two longer wavelength IFUs. There is one 1024 x 1024 detector for each path. Light entering the MRS is spectrally separated into four channels by dichroic mirrors. Each of these channels has its own IFU that divides the image into several slices. Each slice is then dispersed using a grating spectrograph and imaged on one half of a detector. While four channels are observed simultaneously, each exposure only records the spectral coverage of approximately one third of the full wavelength range of each channel. The full 5-28 micron spectrum is obtained by making three exposures using three different gratings and three different dichroic sets. We refer to a sub-channel as one of the three possible configurations (A/B/C) of the channel where each sub-channel covers ~1/3 of the full wavelength range for the channel. Each of the four channels has a different sampling of the field, so the FOV, slice width, number of slices, and plate scales are different for each channel.

The NIRSpec IFU has a 3 x 3 arcsecond field of view that is sliced into thirty 0.1 arcsecond regions. Each slice is dispersed by a prism or one of six diffraction gratings. The NIRSpec IFU gratings provide high-resolution and medium resolution spectroscopy while the prism yields lower-resolution spectroscopy. The NIRSpec detector focal plane consists of two HgCdTe sensor chip assemblies (SCAs). Each SCA is a 2-D array of 2048 x 2048 pixels. For

low or medium resolution IFU data the 30 slices are imaged on a single NIRSpec SCA. In high resolution mode the 30 slices are imaged on the two NIRSpec SCAs.

Terminology

General IFU Terminology

pixel
A pixel is a physical 2-D element of the detector focal plane arrays.

spaxel
A spaxel is a 2-D spatial element of an IFU rectified data cube. Each spaxel in a data cube has an associated spectrum composed of many voxels.

voxel
A voxel is 3-D volume element within an IFU rectified data cube. Each voxel has two spatial dimensions and one spectral dimension.

MIRI Spectral Range Divisions

We use the following terminology to define the spectral range divisions of MIRI:

Channel
The spectral range covered by each MIRI IFU. The channels are labeled as 1, 2, 3 and 4.

Sub-Channel
The 3 sub-ranges that a channel is divided into. These are designated as *Short (A)*, *Medium (B)*, and *Long (C)*.

Band
For **MIRI**, band is one of the 12 contiguous wavelength intervals (four channels times three sub-channels each) into which the spectral range of the MRS is divided. Each band has a unique channel/sub-channel combination. For example, the shortest wavelength range on MIRI is covered by Band 1-SHORT (aka 1A) and the longest is covered by Band 4-LONG (aka 4C).

For **NIRSpec** we define a *band* as a single grating-filter combination, e.g. G140M-F070LP. The possible grating/filter combinations for NIRSpec are given in the table below.

NIRSpec IFU Disperser and Filter Combinations

Grating	Filter	Wavelength (microns)*
Prism	Clear	0.6 -5.3
G140M	F070LP	0.90 - 1.27
G140M	F100LP	0.97 - 1.89
G235M	F170LP	1.66 - 3.17
G395M	F290LP	2.87 - 5.27
G140H	F070LP	0.95 - 1.27
G140H	F100LP	0.97 - 1.89
G235H	F170LP	1.66 - 3.17
G395H	F290LP	2.87 - 5.27

- Approximate wavelength ranges are given to aid in explaining how to build NIRSpec IFU cubes, see NIRSpec Spectral configuration (https://jwst-docs.stsci.edu/jwst-near-infrared-spectrograph/nirspec-observing-modes/nirspec-ifu-spectroscopy#NIRSpecIFUSpectroscopy-Spectralconfigurations).

Types of Output Cubes

The output 3-D spectral data consist of rectangular cube with three orthogonal axes: two spatial and one spectral. Depending on how cube_build is run the spectral axes can be either linear or non-linear. Linear wavelength IFU cubes are constructed from a single band of data, while non-linear wavelength IFU cubes are created from more than one band of data. If the IFU cubes have a non-linear wavelength dimension there will be an added binary extension table to the output fits IFU cube. This extension has the label WCS-TABLE and contains the wavelengths for each of the IFU cube wavelength planes. This table follows the FITs standard described in, *Representations of spectral coordinates in FITS*, Greisen, et al., **A & A** 446, 747-771, 2006.

The input data to `cube_build` can take a variety of forms, including a single file, a data model passed from another pipeline step, a list of files in an association table, or a collection of exposures in a data model container (ModelContainer) passed in by the user or from a preceding pipeline step. Because the MIRI IFUs project data from two channels onto a single detector, choices can or must be made as to which parts of the input data to use when constructing the output cube even in the simplest case of a single input image. The default behavior varies according to the context in which `cube_build` is being run.

In the case of the *calwebb_spec2* pipeline, for example, where the input is a single MIRI or NIRSpec IFU exposure, the default output cube will be built from all the data in that single exposure. For MIRI this means using the data from both channels (e.g. 1A and 2A) that are recorded in a single exposure and the output IFU cube will have a non-linear wavelength dimension. For NIRSpec the data is from the single grating and filter combination contained in the exposure and will have a linear wavelength dimension. The calwebb_spec2 pipeline calls cube_build with `output_type=multi`.

In the *calwebb_spec3* pipeline, on the other hand, where the input can be a collection of data from multiple exposures covering multiple bands, the default behavior is to create a set of single-channel cubes. For MIRI, for example, this can mean separate cubes for channel 1, 2, 3 and 4. depending on what's included in the input. For NIRSpec this may mean multiple cubes, one for each grating+filter combination contained in the input collection. The calwebb_spec3 pipeline calls cube_build with `output_type=band`. These types of IFU cubes will have a linear-wavelength dimension. If the user wants to combine all the data together covering several band they can using the option `output_type=multi` and the resulting IFU cubes will have a non-linear wavelength dimension.

Several `cube_build` step arguments are available to allow the user to control exactly what combinations of input data are used to construct the output cubes. The IFU cubes are constructed, by default, on the sky with north pointing up and east to the left. There are also options to change the output coordinate system, see the *Step Arguments* section for details.

Output Cube Format

The output spectral cubes are stored in FITS files that contain 4 IMAGE extensions. The primary data array is empty and the primary header holds the basic parameters of the observations that went into making the cube. The 4 IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	2 spatial and 1 spectral	float
ERR	3	2 spatial and 1 spectral	float
DQ	3	2 spatial and 1 spectral	integer
WMAP	3	2 spatial and 1 spectral	integer

The SCI image contains the surface brightness of cube spaxels in units of MJy/steradian. The wavelength dimension of the IFU cube can either be linear or non-linear. If the wavelength is non-linear, then the IFU cube contains data from more than one band. A table containing the wavelength of each plane is provided and conforms to the 'WAVE_TAB' fits convention. The wavelengths in the table are read in from the cubepar reference file. The ERR image contains the uncertainty on the SCI values, the DQ image contains the data quality flags for each spaxel, and the WMAP image contains the number of detector pixels contributing to a given voxel. The data quality flag does not propagate the dq flags from previous steps but is defined in the cube build step as: good data (value = 0), non_science (value = 512), do_not_use(value =1), or a combination of non_science and do_not_use (value = 513).

The SCI and ERR cubes are populated with NaN values for voxels where there is no valid data (e.g., outside the IFU cube footprint or for saturated pixels for which no slope could be measured).

Output Product Name

If the input data is passed in as an ImageModel, then the IFU cube will be passed back as an IFUCubeModel. The cube model will be written to disk at the end of processing. The file name of the output cube is based on a rootname plus a string defining the type of IFU cube, along with the suffix 's3d.fits'. If the input data is a single exposure, then the rootname is taken from the input filename. If the input is an association table, the rootname is defined in the association table. The string defining the type of IFU is created according to the following rules:

- For MIRI the output string name is determined from the channels and sub-channels used. The IFU string for MIRI is 'ch'+ channel numbers used plus a string for the subchannel. For example if the IFU cube contains channel 1 and 2 data for the short subchannel, the output name would be, rootname_ch1-2_SHORT_s3d.fits. If all the sub-channels were used then the output name would be rootname_ch-1-2_ALL_s3d.fits.
- For NIRSpec the output string is determined from the gratings and filters used. The gratings are grouped together in a dash (-) separated string and likewise for the filters. For example if the IFU cube contains data from grating G140M and G235M and from filter F070LP and F100LP, the output name would be, rootname_G140M-G225_F070LP-F100LP_s3d.fits

Algorithm

The type of output IFU cube created depends on which pipeline is being run, *calwebb_spec2* or *calwebb_spec3*, and if additional user provided options are being set (see the *Step Arguments* section.). Based on the pipeline setting and any user provided arguments defining the type of cubes to create, the program selects the data from each exposure that should be included in the spectral cube. The output cube is defined using the WCS information of all the input data. The input data are mapped to the output frame based on the wcs information that is filled in by the *assign_wcs* step, this mapping includes any dither offsets. Therefore, the default output cube WCS defines a field-of-view that encompasses the undistorted footprints on the sky of all the input images. The output sampling scale in all three dimensions for the cube is defined by a cubepar reference file as a function of wavelength, and can also be changed by the user. The cubepar reference file contains a predefined scale to use for each dimension for each band. If the output IFU cube contains more than one band, then for MIRI the output scale corresponds to the channel with the smallest scale. In the case of NIRSpec only gratings of the same resolution are combined together in an IFU cube. The default output spatial coordinate system is right ascension-declination. There is an option to create IFU cubes in the coordinate system of the NIRSpec or MIRI MIRS local ifu slicer plane (see *Step Arguments*, coord_system='internal_cal').

The pixels on each exposure that are to be included in the output are mapped to the cube coordinate system. This pixel mapping is determined via a series of chained mapping transformations derived from the WCS of each input image and the WCS of output cube. The mapping process corrects for the optical distortions and uses the spacecraft telemetry information to map each pixel to its projected location in the cube coordinate system.

Weighting

The JWST pipeline includes two methods for building IFU data cubes: the 3D drizzle approach (default), and an alternative based on an exponential modified-Shepard method (EMSM) weighting function. The core principle of both algorithms is to resample the 2-D detector data into a 3D rectified data cube in a single step while conserving flux. The differences in the the techniques are how the detector pixels are weighted in the final 3D data cube.

3-D drizzling

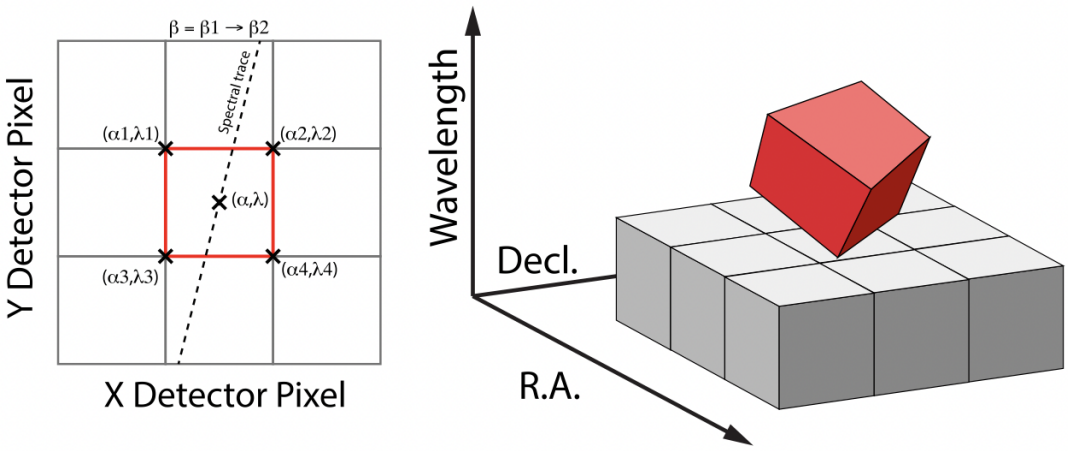
The default method of cube building uses a 3-D drizzling technique analogous to that used by 2-D imaging modes with an additional spectral overlap computation. It is used when `weighting=drizzle`. In the 3D drizzling we project the 2D detector pixels to their corresponding 3D volume elements and allocate their intensities to the individual voxels of the final data cube according to their volumetric overlap. The drizzling algorithm computes the overlap between the irregular projected volumes of the detector pixels and the regular grid of cube voxels, which, for simplicity, we assume corresponds to the world coordinates (R. A., decl.,).

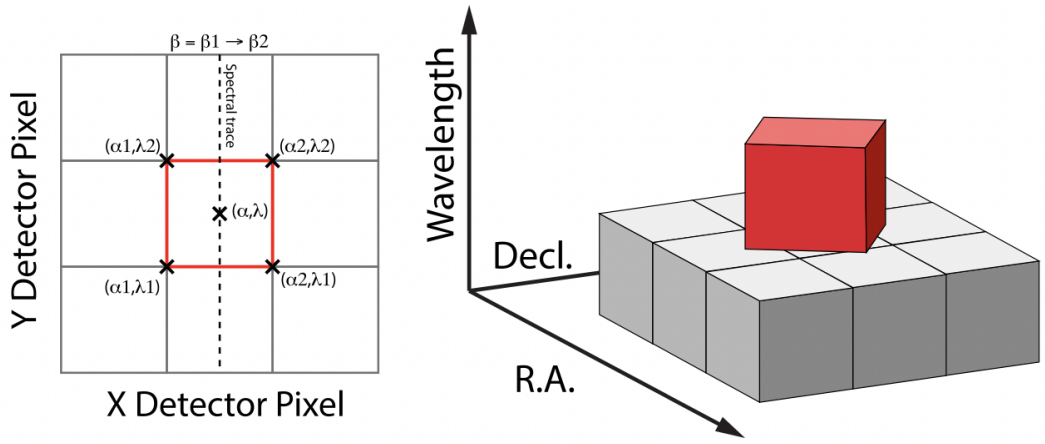
The detector pixels illuminated by JWST slicer-type IFUs contain a mixture of degenerate spatial and spectral information. The spatial extent in the along-slice direction () and the spectral extent in the dispersion direction () both vary continuously within the dispersed image of a given slice in a manner akin to a traditional slit spectrograph and are sampled by the detector pixels (x, y). In contrast, the spatial extent in the across-slice direction () is set by the IFU image slicer width and changes discretely between slices. The four corners of a detector pixel thus define a tilted hexahedron in (,) space with the front and back faces of the polyhedron defined by the lines of constant created by the IFU slicer. (,) is itself rotated (and incorporates some degree of optical distortion) with respect to world coordinates (R.A., Decl.) and thus the volume element defined by a detector pixel is rotated in a complex manner with respect to the cube voxels, see Figure 1. The iso- and iso- directions are not perfectly orthogonal to each other, and are similarly tilted with respect to the detector pixel grid. However, since iso- is nearly aligned with the detector y-axis for MIRI (or x- axis for NIRSpec) and iso- is nearly aligned with the detector x-axis for MIRI (or y-axis for NIRSpec), we make the additional simplifying assumption to ignore this small tilt when computing the projected volume of the detector pixels. Effectively, this means that the surfaces of the volume element are flat in the , , and planes, and the spatial and spectral overlaps can be computed independently (see Figure 2).

With these simplifications, detector pixels project as rectilinear volumes into cube space. The detector pixel flux is redistributed onto a regular output grid according to the relative overlap between the detector pixels and cube voxels. The weighting applied to the detector pixel flux is the product of the fractional spatial and spectral overlap between detector pixels and cube voxels as a function of wavelength. The spatial extent of each detector pixel volume is determined from the combination of the along-slice pixel size and the IFU slice width, both of which will be rotated at some angle with respect to the output voxel grid of the final data cube. The spectral extent of each detector pixel volume is determined by the wavelength range across the pixel in the dimension most closely matched to the dispersion axis (i.e., neglecting small tilts of the dispersion direction with respect to the detector pixel grid). For more details on this method, see 'A 3D Drizzle Algorithm for JWST and Practical Application to the MIRI Medium Resolution Spectrometer', David R. Law et al. 2023 AJ 166 45 (https://iopscience.iop.org/article/10.3847/1538-3881/acdddc).

Figure 1: Left: general case detector diagram in which the dispersion axis is tilted with respect to the detector columns/rows, and the four corners of a given pixel (bold red outline) each have different wavelengths and along-slice coordinates . Right: projection of this generalized detector pixel into the volumetric space of the final data cube. The red hexahedron represents the detector pixel, where the three dimensions are set by the along-slice, across-slice, and wavelength coordinates. The regular gray hexahedra represent voxels in a single wavelength plane of the data cube. For clarity, the cube voxels are shown aligned with the (R.A., Decl.) celestial coordinate frame, but this choice is arbitrary.

Figure 2: Same as Figure 1 but representing the simplified case in which the spectral dispersion is assumed to be aligned with detector columns and the spatial distortion constant for all wavelengths covered by a given pixel. This assumption reduces the computation of volumetric overlap between red and gray hexahedra to separable 1D and 2D computations.


β = β1 → β2
Spectral trace
(α1,λ1)
(α2,λ2)
(α,λ)
(α3,λ3)
(α4,λ4)
Y Detector Pixel
X Detector Pixel
Wavelength
Decl.
R.A.


β = β1 → β2
Spectral trace
(α1,λ2)
(α2,λ2)
(α,λ)
(α1,λ1)
(α2,λ1)
Y Detector Pixel
X Detector Pixel
Wavelength
Decl.
R.A.

Shepard's method of weighting

The second approach to cube building is to use a flux-conserving variant of Shepard's method. In this technique we ignore the overlap between the detector pixel and cube voxel and instead treat each pixel as a single point when mapping the detector to the sky. The mapping process results in an irregularly spaced "cloud of points" that sample the specific intensity distribution at a series of locations on the sky. A schematic of this process is shown in Figure 3.

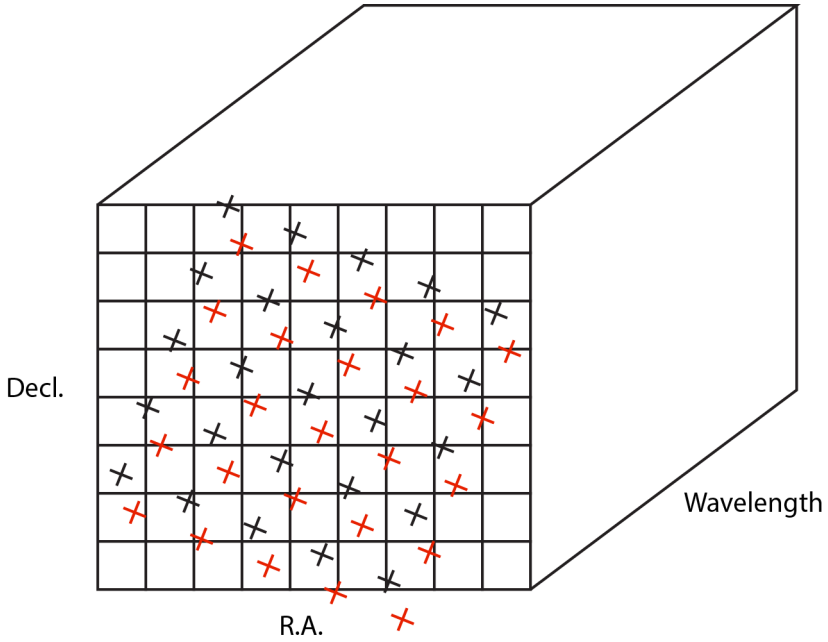


Figure 3: Schematic of two dithered exposures mapped to the IFU output coordinate system (black regular grid). The plus symbols represent the point cloud mapping of detector pixels to effective sampling locations relative to the output coordinate system at a given wavelength. The black points are from exposure one and the red points are from exposure two.

Each point in the cloud represents a measurement of the specific intensity (with corresponding uncertainty) of the astronomical scene at a particular location. The final data cube is constructed by combining each of the irregularly-distributed samples of the scene into a regularly-sampled **voxel** grid in three dimensions for which each **spaxel** (i.e., a spatial pixel in the cube) has a spectrum composed of many spectral elements. The final value of value of a given voxel of the cube is a distance-weighted average of all point-cloud members within a given region of influence.

In order to explain this method we introduce the follow definitions:

- xdistance = distance between point in the cloud and voxel center in units of arc seconds along the x axis
- ydistance = distance between point in the cloud and voxel center in units of arc seconds along the y axis
- zdistance = distance between point in the cloud and voxel center in the lambda dimension in units of microns along the wavelength axis

These distances are then normalized by the IFU cube voxel size for the appropriate axis:

- xnormalized = xdistance/(cube voxel size in x dimension [cdelt1])
- ynormalized = ydistance/(cube voxel size in y dimension [cdelt2])
- znormalized = zdistance/(cube voxel size in z dimension [cdelt3])

The final voxel value at a given wavelength is determined as the weighted sum of the point cloud members with a spatial and spectral region of influence centered on the voxel. The default size of the region of influence is defined in the cubepar reference file, but can be changed by the user with the options: `rois` and `roiw`.

If *n* point cloud members are located within the ROI of a voxel, the voxel flux K = $\frac{\sum_{i=1}^{n} Flux_i w_i}{\sum_{i=1}^{n} w_i}$

where the weighting `weighting=emsm` is:

$w_i = e^{\frac{-(xnormalized_i^2 + ynormalized_i^2 + znormalized_i^2)}{scalefactor}}$

The *scale factor* = *scale rad/cdelt1*, where *scale rad* is read in from the reference file and varies with wavelength.

If the alternative weighting function (set by `weighting = msm`) is selected then:

$w_i = \frac{1.0}{\sqrt{(xnormalized_i^2 + ynormalized_i^2 + znormalized_i^2)^p}}$

In this weighting function the default value for *p* is read in from the cubepar reference file. It can also be set by the argument `weight_power=value`.

Step Arguments

The default values for the step arguments are found in the `CubeBuildStep.spec` attribute. The user can override the default values for a parameter if a step argument exist for the parameter.

The step arguments can be used to control the properties of the output IFU cube or to select subsets of data are used to produce the output cubes. Note that some options will result in multiple cubes being created. For example, if the input data span several bands, but `output_type = band` then a cube for each band will be created.

`channel [string]`
: This is a MIRI only option and the valid values are 1, 2, 3, 4, and ALL. If the `channel` argument is given, then only data corresponding to that channel will be used in constructing the cube. A comma-separated list can be used to designate multiple channels. For example, to create a cube with data from channels 1 and 2, specify the list as `--channel='1,2'`. All the sub-channels (bands) for the chosen channel(s) will be used to create the IFU cube, unless the `band` argument is used to select specific bands. This parameter can be combined with the `output_type` parameter to fully control the type of IFU cubes to make.

`band [string]`
: This is a MIRI only option and the valid values are SHORT, MEDIUM, LONG, and ALL. If the `band` argument is given, then only data corresponding to that sub-channel will be used in constructing the cube. Only one value can be specified. Note we use the name `band` for this argument instead of `subchannel`, because the keyword `band` in the input images is used to indicate which MIRI subchannel the data cover. This parameter can be combined with the `output_type` parameter to fully control the type of IFU cubes to make.

`grating [string]`
: This is a NIRSpec only option with valid values PRISM, G140M, G140H, G235M, G235H, G395M, G395H, and ALL. If the option "ALL" is used, then all the gratings in the association are used. Because association tables only contain exposures of the same resolution, the use of "ALL" will at most combine data from gratings G140M, G235M, and G395M or G140H, G235H, and G395H. The user can supply a comma-separated string containing the names of multiple gratings to use.

`filter [string]`
: This is a NIRSpec only option with values of Clear, F100LP, F070LP, F170LP, F290LP, and ALL. To cover the full wavelength range of NIRSpec, the option "ALL" can be used (provided the exposures in the association table contain all the filters). The user can supply a comma-separated string containing the names of multiple filters to use.

`output_type [string]`
: This parameter has four valid options of Band, Channel, Grating, and Multi. This parameter can be combined with the options above [band, channel, grating, filter] to fully control the type of IFU cubes to make.

- `output_type = band` creates IFU cubes containing only one band (channel/sub-channel for MIRI or grating/filter combination for NIRSpec).

- `output_type = channel` creates a single IFU cube from each unique channel of MIRI data (or just those channels set by the 'channel' option). This is the default mode for the *calwebb_spec3* pipeline for MIRI data.
- `output_type = grating` combines all the gratings in the NIRSpec data or set by the grating option into a single IFU cube. The is the default mode for the *calwebb_spec3* pipeline for NIRSpec data.
- `output_type = multi` combines data into a single "uber" IFU cube, this the default mode for *calwebb_spec2* pipeline. If in addition, channel, band, grating, or filter are also set, then only the data set by those parameters will be combined into an "uber" cube.

The following arguments control the size and sampling characteristics of the output IFU cube.

scalexy
The output cube's spaxel size for axis 1 and 2 (spatial).

scalew
The output cube's spaxel size in axis 3 (wavelength).

wavemin
The minimum wavelength, in microns, to use in constructing the IFU cube.

wavemax
The maximum wavelength, in microns, to use in constructing the IFU cube.

ra_center
Right ascension center, in decimal degrees, of the IFU cube that defines the location of xi/eta tangent plane projection origin.

dec_center
Declination center, in decimal degrees, of the IFU cube that defines the location of xi/eta tangent plane projection origin.

cube_pa
The position angle of the IFU cube in decimal degrees (E from N).

nspax_x
The odd integer number of spaxels to use in the x dimension of the tangent plane.

nspax_y
The odd integer number of spaxels to use in the y dimension of the tangent plane.

coord_system [string]
The default IFU cubes are built on the ra-dec coordinate system (`coord_system=skyalign`). In these cubes north is up and east is left. There are two other coordinate systems an IFU cube can be built on:

- `coord_system=ifualign` is also on the ra-dec system but the IFU cube is aligned with the instrument IFU plane.
- `coord_system=internal_cal` is built on the local internal IFU slicer plane. These types of cubes will be useful during commissioning. For both MIRI ad NIRSpec only a single band from a single exposure can be used to create these type of cubes. The spatial dimensions for these cubes are two orthogonal axes, one parallel and the perpendicular to the slices in the FOV.

There are a number of arguments that control how the point cloud values are combined together to produce the final flux associated with each output spaxel flux. The first set defines the the **region of interest**, which defines the boundary centered on the spaxel center of point cloud members that are used to find the final spaxel flux. The arguments related to region of interest and how the fluxes are combined together are:

rois [float]
The radius of the region of interest in the spatial dimensions.

roiw [float]
The size of the region of interest in the spectral dimension.

weighting [string]
The type of weighting to use when combining detector pixel fluxes to represent the spaxel flux. Allowed values are `emsm`, `msm` and `drizzle`.

For more details on how the weighting of the detector pixel fluxes are used in determining the final spaxel flux see the *Weighting* section.

A parameter only used for investigating which detector pixels contributed to a cube spaxel is `debug_spaxel`. This option is only valid if the `weighting` parameter is set to `drizzle` (default).

`debug_spaxel [string]`

The string is the x,y,z value of the cube spaxel that is being investigated. The numbering starts counting at 0. To print information to the screeen about the x = 10, y = 20, z = 35 spaxel the parameter string value is '10 20 35'.

Examples of How to Run Cube_Build ================================ It is assumed that the input data have been processed through the *calwebb_detector1* pipeline and up through the `photom` step of the *calwebb_spec2* pipeline.

Cube Building for MIRI data

To run cube_build on a single MIRI exposure (containing channel 1 and 2), but only creating an IFU cube for channel 1:

```
strun jwst.cube_build.CubeBuildStep MIRM103-Q0-SHORT_495_cal.fits --ch=1
```

The output 3D spectral cube will be saved in a file called MIRM103-Q0-SHORT_495_ch1-short_s3d.fits

To run cube_build using an association table containing 4 dithered images:

```
strun jwst.cube_build.CubeBuildStep cube_build_4dither_asn.json
```

where the ASN file cube_build_4dither_asn.json contains:

```
{"asn_rule": "Asn_MIRIFU_Dither",
 "target": "MYTarget",
 "asn_id": "c3001",
 "asn_pool": "jw00024_001_01_pool",
 "program": "00024","asn_type":"dither",
 "products": [
             {"name": "MIRM103-Q0-Q3",
             "members":
              [{"exptype": "SCIENCE", "expname": "MIRM103-Q0-SHORT_495_cal.fits"},
               {"exptype": "SCIENCE", "expname": "MIRM103-Q1-SHORT_495_cal.fits"},
               {"exptype": "SCIENCE", "expname": "MIRM103-Q2-SHORT_495_cal.fits"},
               {"exptype": "SCIENCE", "expname": "MIRM103-Q3-SHORT_495_cal.fits"}]}
              ]
}
```

The default output will be two IFU cubes. The first will contain the combined dithered images for channel 1, sub-channel SHORT and the second will contain the channel 2, sub-channel SHORT data. The output root file names are defined by the product "name" attribute in the association table and results in files MIRM103-Q0-Q3_ch1-short_s3d.fits and MIRM103-Q0-Q3_ch2-short_s3d.fits.

To use the same association table, but combine all the data, use the output_type=multi option:

```
strun jwst.cube_build.CubeBuildStep cube_build_4dither_asn.json --output_type=multi
```

The output IFU cube file will be MIRM103-Q0-Q3_ch1-2-short_s3d.fits

Cube building for NIRSpec data

To run `cube_build` on a single NIRSpec exposure that uses grating G140H and filter F100LP:

```
strun jwst.cube_build.CubeBuildStep jwtest1004001_01101_00001_nrs2_cal.fits
```

The output file will be jwtest1004001_01101_00001_nrs2_g140h-f100lp_s3d.fits

To run `cube_build` using an association table containing data from exposures using G140H+F100LP and G140H+F070LP:

```
strun jwst.cube_build.CubeBuildStep nirspec_multi_asn.json
```

where the association file contains:

```
{"asn_rule": "Asn_NIRSPECFU_Dither",
 "target": "MYTarget",
 "asn_pool": "jw00024_001_01_pool",
 "program": "00024","asn_type":"NRSIFU",
 "asn_id":"a3001",
 "products": [
 {"name": "JW3-6-NIRSPEC",
 "members":
 [{"exptype": "SCIENCE", "expname": "jwtest1003001_01101_00001_nrs1_cal.fits"},
 {"exptype": "SCIENCE", "expname": "jwtest1004001_01101_00001_nrs2_cal.fits"},
 {"exptype": "SCIENCE", "expname": "jwtest1005001_01101_00001_nrs1_cal.fits"},
 {"exptype": "SCIENCE", "expname": "jwtest1006001_01101_00001_nrs2_cal.fits"}]}
 ]
 }
```

The output will be two IFU cubes, one for each grating+filter combination: JW3-6-NIRSPEC_g140h-f070lp_s3d.fits and JW3-6-NIRSPEC_g140h-f100lp_s3d.fits.

Reference Files

The `cube_build` step uses CUBEPAR reference file.

CUBEPAR reference file

REFTYPE
CUBEPAR

Data models
`MiriIFUCubeParsModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MiriIFUCubeParsModel.html#
`NirspecIFUCubeParsModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecIFUCubeParsMode

The CUBEPAR reference file contains parameter values used to construct the output IFU cubes.

Reference Selection Keywords for CUBEPAR

CRDS selects appropriate CUBEPAR references based on the following keywords. CUBEPAR is not applicable for instruments not in the table.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSPEC	INSTRUME, EXP_TYPE, OPMODE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for CUBEPAR

In addition to the standard reference file keywords listed above, the following keywords are *required* in CUBEPAR reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for CUBEPAR*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

MIRI Reference File Format

The MIRI CUBEPAR reference files are FITS format, with 5 BINTABLE extensions. The FITS primary data array is assumed to be empty. The format and content of the MIRI CUBEPAR reference file

EXTNAME	XTENSION	Dimensions
CUBEPAR	BINTABLE	TFIELDS = 6
CUBEPAR_MSM	BINTABLE	TFIELDS = 6
MULTICHANNEL_MSM	BINTABLE	TFIELDS = 5
CUBEPAR_EMSM	BINTABLE	TFIELDS = 5
MULTICHANNEL_EMSM	BINTABLE	TFIELDS = 4
MULTICHANNEL_DRIZ	BINTABLE	TFIELDS = 1

NIRSPec Reference File Format

The NIRSpec CUBEPAR reference files are FITS format, with 9 BINTABLE extensions.

EXTNAME	XTENSION	Dimensions
CUBEPAR	BINTABLE	TFIELDS = 6
CUBEPAR_MSM	BINTABLE	TFIELDS = 6
MULTICHAN_PRISM_MSM	BINTABLE	TFIELDS = 5
MULTICHAN_MED_MSM	BINTABLE	TFIELDS = 5
MULTICHAN_HIGH_MSM	BINTABLE	TFIELDS = 5
CUBEPAR_EMSM	BINTABLE	TFIELDS = 5
MULTICHAN_PRISM_EMSM	BINTABLE	TFIELDS = 4
MULTICHAN_MED_EMSM	BINTABLE	TFIELDS = 4
MULTICHAN_HIGH_EMSM	BINTABLE	TFIELDS = 4

The formats of the individual table extensions are listed below, first for the MIRI reference file and then for NIRSpec.

Table	Column	Data type	Units
CUBEPAR	CHANNEL	shortint	N/A
	BAND	ch*6	N/A
	WAVEMIN	float	microns
	WAVEMAX	float	microns
	SPAXELSIZE	float	arcseconds
	SPECTRALSTEP	double	microns
CUBEPAR_MSM	CHANNEL	shortint	N/A
	BAND	ch*6	N/A
	ROISPATIAL	float	arcseconds
	ROISPECTRAL	double	microns
	POWER	double	unitless
	SOFTRAD	double	unitless
MULTICHANNEL_MSM	WAVELENGTH	double	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	POWER	double	unitless
	SOFTRAD	double	unitless
CUBEPAR_EMSM	CHANNEL	shortint	N/A
	BAND	ch*6	N/A
	ROISPATIAL	float	arcseconds
	ROISPECTRAL	double	microns
	SCALERAD	double	unitless
MULTICHANNEL_EMSM	WAVELENGTH	double	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	SCALERAD	double	unitless
MULTICHANNEL_DRIZ	WAVELENGTH	double	microns

Table	Column	Data type	Units
CUBEPAR	DISPERSER	ch*5	N/A
	FILTER	ch*6	N/A
	WAVEMIN	double	microns
	WAVEMAX	double	microns
	SPAXELSIZE	double	arcseconds
	SPECTRALSTEP	double	microns
CUBEPAR_MSM	DISPERSER	ch*5	N/A
	FILTER	ch*6	N/A
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	POWER	double	unitless
	SOFTRAD	double	unitless
MULTI-CHAN_PRISM_MSM	WAVELENGTH	double	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	POWER	double	unitless
	SOFTRAD	double	unitless
MULTI-CHAN_MED_MSM	WAVELENGTH	float	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns

continues on next page

Table 2 – continued from previous page

Table	Column	Data type	Units
	POWER	double	unitless
	SOFTRAD	double	unitless
MULTI-CHAN_HIGH_MSM	WAVELENGTH	float	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	POWER	double	unitless
	SOFTRAD	double	unitless
CUBEPAR_EMSM	DISPERSER	ch*5	N/A
	FILTER	ch*6	N/A
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	SCALERAD	double	unitless
MULTI-CHAN_PRISM_EMSM	WAVELENGTH	double	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	SCALERAD	double	unitless
MULTI-CHAN_MED_EMSM	WAVELENGTH	float	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	SCALERAD	double	unitless
MULTI-CHAN_HIGH_EMSM	WAVELENGTH	float	microns
	ROISPATIAL	double	arcseconds
	ROISPECTRAL	double	microns
	SCALERAD	double	unitless

These reference files contain tables for each wavelength band giving the spatial and spectral size, and the size of the region of interest (ROI) to use to construct an IFU cube. If only one band is used to construct the IFU cube then the *CUBEPAR* and *CUBEPAR_MSM* or *CUBE_EMSM* tables are used. These types of cubes will have a linear - wavelength dimension. If more than one wavelength band is used to build the IFU cube then the MULTICHANNEL (MIRI) or MULTICHAN (NIRSPEC) tables are used o set the spectral and spatial roi size, and the wavelength dependent weighting function parameters. For multi-band IFU cubes then the final spatial size will be the smallest one from the list of input bands and these cubes will have a non-linear wavelength dimension.

The MIRI reference table descriptions:

- **CUBEPAR** table contains the spatial and spectral cube sample size for each band.
- **CUBEPAR_MSM** table contains the Modified Shepard Method (MSM) weighting values to use for each band.
- **MULTICHANNEL_MSM** table is used for the MSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from several bands and the final output is to have an IFU cube of varying spectral scale.
- **CUBEPAR_EMSM** table contains the Exponential Modified Shepard Method (EMSM) weighting values to use for each band.
- **MULTICHANNEL_EMSM** table is used for the EMSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from several bands and the final output is to have an IFU cube of varying spectral scale.
- **MULTICHANNEL_DRIZ** table is used for the DRIZZLE weighting and contains the wavelengths to use when IFU cubes are created from several bands and the final output is to have an IFU cube of varying spectral scale.

The NIRSPEC reference table descriptions:

- **CUBEPAR** table contains the spatial and spectral cube sample size for each band.
- **CUBEPAR_MSM** table contains the Modified Shepard Method (MSM) weighting values to use for each band.
- **MULTICHAN_PRISM_MSM** table is used for the MSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from the grating prism and the final IFU Cube output has a varying spectral scale.
- **MULTICHAN_MED_MSM** table is used for the MSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from the medium resolution grating and the final IFU Cube output has a varying spectral scale.
- **MULTICHAN_HIGH_MSM** table is used for the MSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from the high resolution gratings and the final IFU Cube output has a varying spectral scale.
- **CUBEPAR_EMSM** table contains the Exponential Modified Shepard Method (EMSM) weighting values to use for each band.
- **MULTICHAN_PRISM_EMSM** table is used for the EMSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from the grating prism and the final IFU Cube output has a varying spectral scale.
- **MULTICHAN_MED_EMSM** table is used for the EMSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from the medium resolution grating and the final IFU Cube output has a varying spectral scale.
- **MULTICHAN_HIGH_EMSM** table is used for the EMSM weighting and contains the wavelengths and associated region of interest size to use when IFU cubes are created from the high resolution gratings and the final IFU Cube output has a varying spectral scale.

jwst.cube_build.cube_build_step Module

This is the main ifu spectral cube building routine.

Classes

CubeBuildStep([name, parent, config_file, ...])	CubeBuildStep: Creates a 3-D spectral cube

CubeBuildStep

class jwst.cube_build.cube_build_step.**CubeBuildStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

CubeBuildStep: Creates a 3-D spectral cube

Notes

This is the controlling routine for building IFU Spectral Cubes. It loads and sets the various input data and parameters need by the cube_build_step.

This routine does the following operations:

1. Extracts the input parameters from the cubepars reference file and merges them with any user-provided values. 2. Creates the output WCS from the input images and defines the mapping between all the input arrays and the output array 3. Passes the input data to the function to map all thei input data to the output array. 4. Updates the output data model with correct meta data

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input)	This is the main routine for IFU spectral cube building.
read_user_input()	Read user input options for channel, subchannel, filter, or grating

Attributes Documentation

class_alias = 'cube_build'

reference_file_types = ['cubepar']

spec

```
channel = option('1','2','3','4','all',default='all') # Channel
band = option('short','medium','long','short-medium','short-long','medium-short
↪',                 'medium-long', 'long-short', 'long-medium','all',default=
↪'all') # Band
grating   = option('prism','g140m','g140h','g235m','g235h','g395m','g395h','all
↪',default='all') # Grating
filter    = option('clear','f100lp','f070lp','f170lp','f290lp','all',default='all
↪') # Filter
output_type = option('band','channel','grating','multi',default=None) # Type␣
↪IFUcube to create.
scalexy = float(default=0.0) # cube sample size to use for axis 1 and axis2,␣
↪arc seconds
scalew = float(default=0.0) # cube sample size to use for axis 3, microns
weighting = option('emsm','msm','drizzle',default = 'drizzle') # Type of␣
↪weighting function
coord_system = option('skyalign','world','internal_cal','ifualign',default=
↪'skyalign') # Output Coordinate system.
ra_center = float(default=None) # RA center of the IFU cube
dec_center = float(default=None) # Declination center of the IFU cube
cube_pa = float(default=None) # The position angle of the desired cube in␣
↪decimal degrees E from N
nspax_x = integer(default=None) # The odd integer number of spaxels to use in␣
↪the x dimension of cube tangent plane.
nspax_y = integer(default=None) # The odd integer number of spaxels to use in␣
↪the y dimension of cube tangent plane.
rois = float(default=0.0) # region of interest spatial size, arc seconds
roiw = float(default=0.0) # region of interest wavelength size, microns
weight_power = float(default=2.0) # Weighting option to use for Modified␣
↪Shepard Method
wavemin = float(default=None)  # Minimum wavelength to be used in the IFUCube
wavemax = float(default=None)  # Maximum wavelength to be used in the IFUCube
single = boolean(default=false) # Internal pipeline option used by mrs_imatch &␣
↪outlier detection
skip_dqflagging = boolean(default=false) # skip setting the DQ plane of the IFU
search_output_file = boolean(default=false)
output_use_model = boolean(default=true) # Use filenames in the output models
suffix = string(default='s3d')
debug_spaxel = string(default='-1 -1 -1') # Default not used
```

Methods Documentation

process(*input*)

This is the main routine for IFU spectral cube building.

Parameters

input (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of objects or str* (https://docs.python.org/3/library/stdtypes.html#str)) – list of datamodels or string name of input fits file or association.

read_user_input()

Read user input options for channel, subchannel, filter, or grating

Class Inheritance Diagram

15.1.14 Dark Current Subtraction

Description

Class

jwst.dark_current.DarkCurrentStep

Alias

dark_current

Assumptions

It is assumed that the input science data have *NOT* had the zero group (or bias) subtracted. We also do not want the dark subtraction process to remove the bias signal from the science exposure, therefore the dark reference data should have their own group zero subtracted from all groups. This means that group zero of the dark reference data will effectively be zero-valued.

Algorithm

The algorithm for this step is called from the external package `stcal`, an STScI effort to unify common calibration processing algorithms for use by multiple observatories.

The dark current step removes dark current from an exposure by subtracting dark current data stored in a dark reference file in CRDS.

The current implementation uses dark reference files that have been constructed from exposures using NFRAMES=1 and GROUPGAP=0 (i.e. one frame per group and no dropped frames) and the maximum number of frames allowed for an integration. If the science exposure that's being processed also used NFRAMES=1 and GROUPGAP=0, then the dark reference file data are directly subtracted group-by-group from the science exposure.

If the science exposure used NFRAMES>1 or GROUPGAP>0, the dark reference file data are reconstructed on-the-fly by the step to match the frame averaging and groupgap settings of the science exposure. The reconstructed dark data are created by averaging NFRAMES adjacent dark frames and skipping GROUPGAP intervening frames.

The frame-averaged dark is constructed using the following scheme:

1. SCI arrays are computed as the mean of the original dark SCI arrays
2. ERR arrays are computed as the uncertainty in the mean, using $\frac{\sqrt{\sum \mathrm{ERR}^2}}{nframes}$

The dark reference data are not integration-dependent for most instruments, hence the same group-by-group dark current data are subtracted from every integration of the science exposure. An exception to this rule is the JWST MIRI instrument, for which the dark signal **is** integration-dependent, at least to a certain extent. MIRI dark reference file data is therefore 4-dimensional (ncols x nrows x ngroups x nintegrations). Typical MIRI dark reference files contain data for only 2 or 3 integrations, which are directly subtracted from the corresponding first few integrations of the science exposure. The data in the last integration of the dark reference file is applied to all remaining science integrations.

The ERR arrays of the science data are currently not modified by this step.

The DQ flags from the dark reference file are propagated into the science exposure PIXELDQ array using a bitwise OR operation.

Upon successful completion of the dark subtraction the S_DARK keyword is set to "COMPLETE".

Special Handling

Any pixel values in the dark reference data that are set to NaN will have their values reset to zero before being subtracted from the science data, which will effectively skip the dark subtraction operation for those pixels.

Note: If the input science exposure contains more groups than the available dark reference file, no dark subtraction will be applied and the input data will be returned unchanged.

Subarrays

It is assumed that dark current will be subarray-dependent, therefore this step makes no attempt to extract subarrays from the dark reference file to match input subarrays. It instead relies on the presence of matching subarray dark reference files in CRDS.

JWST/NIRCam Target Acq Subarrays

Due to the very large number of available NIRCam target acquisition (TA) subarrays, the instrument team has chosen to not provide dark reference files for any of the TA subarrays in CRDS. Requests from the calibration pipeline to CRDS for matching dark reference files to use when processing a NIRCam TA will result in a reference file name of "N/A" being returned, which causes the dark subtraction step to skip processing. Hence dark current will not be subtracted from NIRCam TA subarray exposures.

Step Arguments

The dark current step has one step-specific argument:

- `--dark_output`

If the `dark_output` argument is given with a filename for its value, the frame-averaged dark data that are created within the step will be saved to that file.

Reference File

The `dark` step uses a DARK reference file.

DARK Reference File

REFTYPE
DARK

Data models
`DarkModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.DarkModel.html#jwst.datamodels.DarkMode
`DarkMIRIModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.DarkMIRIModel.html#jwst.datamodels.

The DARK reference file contains pixel-by-pixel and frame-by-frame dark current values for a given detector readout mode.

Reference Selection Keywords for DARK

CRDS selects appropriate DARK references based on the following keywords. DARK is not applicable for instruments not in the table.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, READPATT, SUBARRAY, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, READPATT, SUBARRAY, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, EXP_TYPE, NOUTPUTS, SUBARRAY, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, READPATT, SUBARRAY, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, READPATT, SUBARRAY, SUBSTRT1, SUBSTRT2, SUBSIZE1, SUBSIZE2, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for DARK

In addition to the standard reference file keywords listed above, the following keywords are *required* in DARK reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for DARK*):

Keyword	Data Model Name	Instruments
DETECTOR	model.meta.instrument.detector	All
EXP_TYPE	model.meta.exposure.type	NIRCam
NOUTPUTS	model.meta.exposure.noutputs	NIRCam
READPATT	model.meta.exposure.readpatt	FGS, MIRI, NIRISS, NIRSpec
SUBARRAY	model.meta.subarray.name	All
SUBSTRT1	model.meta.subarray.xstart	NIRSpec
SUBSTRT2	model.meta.subarray.ystart	NIRSpec
SUBSIZE1	model.meta.subarray.xsize	NIRSpec
SUBSIZE2	model.meta.subarray.ysize	NIRSpec

Reference File Format

DARK reference files are FITS format, with 3 IMAGE extensions and 1 BINTABLE extension. The FITS primary data array is assumed to be empty. The format and content of the files is different for MIRI than the near-IR instruments, as shown below.

Near-IR Detectors

Characteristics of the three IMAGE extensions for DARK files used with the Near-IR instruments are as follows (see DarkModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.DarkModel.html#jwst.datamodels.DarkModel)):

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x ngroups	float
ERR	3	ncols x nrows x ngroups	float
DQ	2	ncols x nrows	integer
DQ_DEF	2	TFIELDS = 4	N/A

MIRI Detectors

The DARK reference files for the MIRI detectors depend on the integration number, because the first integration of MIRI exposures contains effects from the detector reset and are slightly different from subsequent integrations. Currently the MIRI DARK reference files contain a correction for only two integrations: the first integration of the DARK is subtracted from the first integration of the science data, while the second DARK integration is subtracted from all subsequent science integrations. The format of the MIRI DARK reference files is as follows (see DarkMIRIModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.DarkMIRIModel.html#jwst.datamodels.DarkMIRIModel)):

EXTNAME	NAXIS	Dimensions	Data type
SCI	4	ncols x nrows x ngroups x nints	float
ERR	4	ncols x nrows x ngroups x nints	float
DQ	4	ncols x nrows x 1 x nints	integer
DQ_DEF	2	TFIELDS = 4	N/A

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

jwst.dark_current Package

Classes

DarkCurrentStep([name, parent, config_file, ...])	DarkCurrentStep: Performs dark current correction by subtracting dark current reference data from the input science data model.

DarkCurrentStep

class jwst.dark_current.**DarkCurrentStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

DarkCurrentStep: Performs dark current correction by subtracting dark current reference data from the input science data model.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'dark_current'

reference_file_types = ['dark']

spec

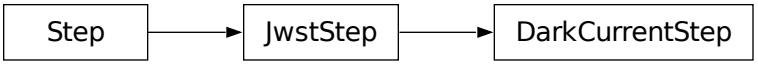
```
dark_output = output_file(default = None) # Dark model or averaged dark␣
↪subtracted
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.15 Data Quality (DQ) Initialization

Description

Class

jwst.dq_init.DQInitStep

Alias

dq_init

The Data Quality (DQ) initialization step in the calibration pipeline populates the DQ mask for the input dataset. Flag values from the appropriate static mask ("MASK") reference file in CRDS are copied into the "PIXELDQ" array of the input dataset, because it is assumed that flags in the mask reference file pertain to problem conditions that affect all groups and integrations for a given pixel.

The actual process consists of the following steps:

1. Determine what MASK reference file to use via the interface to the bestref utility in CRDS.
2. If the "PIXELDQ" or "GROUPDQ" arrays of the input dataset do not already exist, which is sometimes the case for raw input products, create these arrays in the input data model and initialize them to zero. The "PIXELDQ" array will be 2D, with the same number of rows and columns as the input science data. The "GROUPDQ" array will be 4D with the same dimensions (nints, ngroups, nrows, ncols) as the input science data array.
3. Check to see if the input science data is in subarray mode. If so, extract a matching subarray from the full-frame MASK reference file.
4. Propagate the DQ flags from the reference file DQ array to the science data "PIXELDQ" array using numpy's `bitwise_or` function.

Note that when applying the `dq_init` step to FGS guide star data, as is done in the *calwebb_guider* pipeline, the flags from the MASK reference file are propagated into the guide star dataset "DQ" array, instead of the "PIXELDQ" array. The step identifies guide star data based on the following exposure type (EXP_TYPE keyword) values: FGS_ID-IMAGE, FGS_ID-STACK, FGS_ACQ1, FGS_ACQ2, FGS_TRACK, and FGS_FINEGUIDE.

NIRSpec IRS2

No special handling is required for NIRSpec exposures taken using the IRS2 readout pattern, because matching IRS2 MASK reference files are supplied in CRDS.

Step Arguments

The Data Quality Initialization step has no step-specific arguments.

Reference Files

The `dq_init` step uses a MASK reference file.

MASK Reference File

REFTYPE
MASK

Data model
`MaskModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MaskModel.html#jwst.datamodels.MaskMod

The MASK reference file contains pixel-by-pixel DQ flag values that indicate problem conditions.

Reference Selection Keywords for MASK

CRDS selects appropriate MASK references based on the following keywords. MASK is not applicable for instruments not in the table.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, SUBARRAY, EXP_TYPE, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, SUBARRAY, READPATT, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for MASK

In addition to the standard reference file keywords listed above, the following keywords are *required* in MASK reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for MASK*):

Keyword	Data Model Name	Instruments
DETECTOR	model.meta.instrument.detector	All
SUBARRAY	model.meta.subarray.name	All
EXP_TYPE	model.meta.exposure.type	FGS only
READPATT	model.meta.exposure.readpatt	NIRSpec only

Reference File Format

MASK reference files are FITS format, with one IMAGE extension and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
DQ	IMAGE	2	ncols x nrows	integer
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The values in the `DQ` array give the per-pixel flag conditions that are to be propagated into the science exposure's `PIXELDQ` array. The dimensions of the `DQ` array should be equal to the number of columns and rows in a full-frame readout of a given detector, including reference pixels. Note that this does not include the reference *output* for MIRI detectors.

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

jwst.dq_init Package

Classes

DQInitStep([name, parent, config_file, ...])	Initialize the Data Quality extension from the mask reference file.

DQInitStep

class jwst.dq_init.**DQInitStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Initialize the Data Quality extension from the mask reference file.

The dq_init step initializes the pixeldq attribute of the input datamodel using the MASK reference file. For some FGS exp_types, initialize the dq attribute of the input model instead. The dq attribute of the MASK model is bitwise OR'd with the pixeldq (or dq) attribute of the input model.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types

Methods Summary

process(input)	Perform the dq_init calibration step

Attributes Documentation

class_alias = 'dq_init'

reference_file_types = ['mask']

Methods Documentation

process(*input*)

Perform the dq_init calibration step

Parameters
input (*JWST datamodel*) – input jwst datamodel

Returns
output_model – result JWST datamodel

Return type
JWST datamodel

Class Inheritance Diagram

15.1.16 MIRI EMI Correction

Description

Class
jwst.emicorr.EmiCorrStep

Alias
emicorr

Overview

The `emicorr` step corrects for known noise patterns in the raw MIRI data. The majority of the MIRI subarrays have an 390 Hz or other electromagnetic interference (EMI) noise pattern in the raw data. The known frequencies to correct for are in the EMI reference file, under the key `frequencies`. The effect of these EMI frequencies is to imprint each into the rate images. For short integrations in `LRSSLITLESS` the correlated noise from this is quite apparent in the rate images. For longer integrations the net effect is to increase the read noise by about 20%.

The process to do the correction is the following (repeated recursively for each discrete EMI frequency desired):

1. Read image data.
2. Make very crude slope image and fixed pattern "super" bias for each integration, ignoring everything (nonlin, saturation, badpix, etc).
3. Subtract scaled slope image and bias from each frame of each integration.
4. Calculate phase of every pixel in the image at the desired EMI frequency (e.g. 390 Hz) relative to the first pixel in the image.
5. Make a binned, phased amplitude (pa) wave from the cleaned data (plot cleaned vs phase, then bin by phase).
6. Measure the phase shift between the binned pa wave and the input reference wave.[1]
7. Use look-up table to get the aligned reference wave value for each pixel (the noise array corresponding to the input image).[Page 246, 1]
8. Subtract the noise array from the input image and return the cleaned result.

The long term plan is a change to the sizes and locations of the subarrays to get the frame times to be in phase with the known noise frequencies like the full frame images. For the previous and near term observations this can be fixed through application of the `emicorr` step.

An EMICORR reference file can be used to correct for all known noise patterns. The reference file is expected to be in ASDF format, containing the exact frequency numbers, the corresponding 1D array for the phase amplitudes, and a `subarray_cases` dictionary that contains the frequencies to correct for according to subarray, read pattern, and detector. If there is no reference file found in CRDS, the step has a set of default frequencies and subarray cases for which the correction is applied.

Input

The input file is the `_uncal` file after the *dq_init_step* step has been applied, in the in the *calwebb_detector1* pipeline.

Output

The output will be a partially-processed `RampModel` with the corrected data in the SCI extension, meaning, the effect of the EMI frequencies (either the default values or the ones in the reference file) removed. All other extensions will remain the same.

[1] Alternately, use the binned pa wave instead of the reference wave to "self-correct".

Step Arguments

The `emicorr` step has the following step-specific arguments.

`--nints_to_phase` (integer, default=None)
Number of integrations to phase.

`--nbins` (integer, default=None)
Number of bins in one phased wave.

`--scale_reference` (boolean, default=True)
If True, the reference wavelength will be scaled to the data's phase amplitude.

`--user_supplied_reffile` (boolean, default=None)
This is to specify an ASDF-format user-created reference file.

`--save_intermediate_results` (string, default=False)
This is a boolean flag to specify whether to write a step output file with the EMI correction. Additionaly, if the flag `user_supplied_reffile` is None and no CRDS reference file was found, all the on-the-fly frequencies phase amplitudes will be saved to an ASDF output with the same format as an EMI reference file.

Reference Files

The `emicorr` step can use an EMI reference file.

EMICORR Reference File

REFTYPE
EMICORR

Data model
`EmiModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.EmiModel.html#jwst.datamodels.EmiModel)

The EMICORR reference file contains data necessary for removing contaminating MIRI EMI frequencies.

Reference Selection Keywords for EMICORR

CRDS selects appropriate EMICORR references based on the following keywords. EMICORR is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, DATE-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

EMICORR Reference File Format

MIRI EMICORR reference files are in ASDF format. The EMICORR reference file contains the frequencies for which the image will be corrected. Example file contents:

```
root (AsdfObject)
  ├─frequencies (dict)
  │   ├─Hz10 (dict)
  │   │  ├─frequency (float): 10.039216
  │   │  └─phase_amplitudes (NDArrayType): shape=(500,), dtype=float32
  │   ├─Hz10_slow_MIRIFULONG (dict)
  │   │  ├─frequency (float): 10.039216
  │   │  └─phase_amplitudes (NDArrayType): shape=(250,), dtype=float32
  └─subarray_cases (dict)
    ├─BRIGHTSKY (dict)
    │  ├─frameclocks (int): 23968
    │  ├─freqs (dict)
    │  │  ├─FAST (list)
    │  │  │  └─[0] (str): Hz10
    │  │  └─SLOW (dict)
    │  │     ├─MIRIFULONG (list)
    │  │     │  └─[0] (str): Hz10_slow_MIRIFULONG
    │  │     ├─MIRIFUSHORT (list)
    │  │     │  └─[0] (str): Hz10_slow_MIRIFUSHORT
    │  │     └─MIRIMAGE (list)
    │  │        └─[0] (str): Hz10_slow_MIRIMAGE
    │  └─rowclocks (int): 82
    ├─MASK1140 (dict)
    │  ├─frameclocks (int): 23968
    │  ├─freqs (dict)
```

(continues on next page)

(continued from previous page)

```
    │   ├─FAST (list)
    │   │ ├─[0] (str): Hz390
    │   │ └─[1] (str): Hz10
    │   └─SLOW (dict)
    │     ├─MIRIFULONG (list)
    │     │ ├─[0] (str): Hz390
    │     │ └─[1] (str): Hz10_slow_MIRIFULONG
    │     ├─MIRIFUSHORT (list)
    │     │ ├─[0] (str): Hz390
    │     │ └─[1] (str): Hz10_slow_MIRIFUSHORT
    │     └─MIRIMAGE (list)
    │       ├─[0] (str): Hz390
    └─rowclocks (int): 82
```

Frequency Selection

The frequency to be corrected for will be selected according to the dictionary contained in the key `subarray_cases` in the reference file. This contains the subarray names and the names of the corresponding frequencies to be used in the correction.

jwst.emicorr Package

Classes

EmiCorrStep([name, parent, config_file, ...])	EmiCorrStep: Apply MIRI EMI correction to a science image.

EmiCorrStep

class jwst.emicorr.**EmiCorrStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

EmiCorrStep: Apply MIRI EMI correction to a science image.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'emicorr'

reference_file_types = ['emicorr']

spec

```
save_intermediate_results = boolean(default=False)
user_supplied_reffile = string(default=None)  # ASDF user-supplied reference␣
↪file
nints_to_phase = integer(default=None)  # Number of integrations to phase
nbins = integer(default=None)  # Number of bins in one phased wave
scale_reference = boolean(default=True)  # If True, the reference wavelength␣
↪will be scaled to the data's phase amplitude
skip = boolean(default=True)
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.17 Exposure to Source Conversion

Description

`exp_to_source` is a data reorganization tool that is used to convert Stage 2 exposure-based data products to Stage 3 source-based data products. It is only used when there is a known source list for the exposure data, which is required in order to reorganize the data by source. Hence it is only useable for NIRSpec MOS, NIRSpec fixed-slit, NIRCam WFSS, and NIRISS WFSS data. Details on the operation for each mode are given below.

The tool is run near the beginning of the *calwebb_spec3* pipeline, immediately after the *master background* step.

In general, the tool takes as input multiple exposure-based "cal" products created during Stage 2 spectroscopic (*calwebb_spec2*) processing and reorganizes the data in them to create a set of output source-based "cal" products, which are then processed through the remaining steps of the *calwebb_spec3* pipeline. For example, if the input consists of a set of 3 exposure-based "cal" files (from a 3-point nod dither pattern, for example), each one of which contains data for 5 defined sources, then the output consists of a set of 5 source-based "cal" products (one per source), each of which contains the data from the 3 exposures for each source. This is done as a convenience, in order to have all the data for a given source contained in a single product. All data arrays associated with a given source, e.g. SCI, ERR, DQ, WAVELENGTH, VAR_POISSON, etc., are copied from each input product into the corresponding output product.

NIRSpec MOS

NIRSpec MOS observations are created at the APT level by defining a configuration of MSA slitlets with a source assigned to each slitlet. The source-to-slitlet linkage is carried along in the information contained in the MSA metadata file used during *calwebb_spec2* processing. Each slitlet instance created by the *extract_2d* step stores the source ID (a simple integer number) in the SOURCEID keyword of the SCI extension header for the slitlet. The `exp_to_source` tool uses the SOURCEID values to sort the data from each input product into an appropriate source-based output product.

NIRSpec Fixed-Slit

NIRSpec fixed-slit observations do not have sources identified with each slit, so the slit names, e.g. S200A1, S1600A1, etc., are mapped to predefined source ID values, as follows:

Slit Name	Source ID
S200A1	1
S200A2	2
S400A1	3
S1600A1	4
S200B1	5

The assigned source ID values are used by `exp_to_source` to sort the data from each slit into the appropriate source-based output product.

NIRCam and NIRISS WFSS

Wide-Field Slitless Spectroscopy (WFSS) modes do not have a predefined source list, but a source list is created by the *calwebb_image3* pipeline when it processes the direct image exposures that are included in a WFSS observation. That source catalog is then used during *calwebb_spec2* processing of the grism exposures to define and create cutouts for each identified source. Like NIRSpec MOS mode, each "slit" instance is identified by the source ID number from the catalog and is used by the `exp_to_source` tool to reorganize exposures into source-based products.

File Name Syntax

The input exposure-based "cal" products have file names that follow the standard Stage 2 exposure syntax, such as:

```
jw93065002001_02101_00001_nrs1_cal.fits
```

See *exposure-based file names* for more details on the meaning of each field in the file names.

The FITS file naming scheme for the source-based "cal" products follows the standard Stage 3 syntax, such as:

```
jw10006-o010_s00061_nirspec_f170lp-g235m_cal.fits
```

where "s00061" in this example is the source ID. See *source-based file names* for more details on the meaning of each field in this type of file name.

jwst.exp_to_source Package

Functions

exp_to_source(inputs)	Reformat exposure-based MSA data to source-based.
multislit_to_container(inputs)	Reformat exposure-based MSA data to source-based containers.

exp_to_source

jwst.exp_to_source.**exp_to_source**(*inputs*)

Reformat exposure-based MSA data to source-based.

Parameters

inputs ([*MultiSlitModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jw
...]) – List of MultiSlitModel instances to reformat.

Returns

multiexposures – Returns a dict of MultiExposureModel instances wherein each instance contains slits belonging to the same source. The key is the ID of each source, i.e. `source_id`.

Return type

dict (https://docs.python.org/3/library/stdtypes.html#dict)

multislit_to_container

jwst.exp_to_source.**multislit_to_container**(*inputs*)

Reformat exposure-based MSA data to source-based containers.

Parameters

inputs ([*MultiSlitModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jw
...]) – List of MultiSlitModel instances to reformat, or just a ModelContainer full of MultiSlitModels.

Returns

containers – Returns a dict of ModelContainer instances wherein each instance contains ImageModels of slits belonging to the same source. The key is the ID of each slit, i.e. `source_id`.

Return type

dict (https://docs.python.org/3/library/stdtypes.html#dict)

15.1.18 Extract 1D Spectra

Description

Class

jwst.extract_1d.Extract1dStep

Alias

extract_1d

Overview

The `extract_1d` step extracts a 1D signal from a 2D or 3D dataset and writes spectral data to an "x1d" product. This works on all JWST spectroscopic modes, including MIRI LRS (slit and slitless) and MRS, NIRCam WFSS and TSGRISM, NIRISS WFSS and SOSS, and NIRSpec fixed-slit, IFU, and MOS.

An EXTRACT1D reference file is used for most modes to specify the location and size of the target and background extraction apertures. The EXTRACT1D reference file is not used for Wide-Field Slitless Spectroscopy data (NIS_WFSS or NRC_WFSS). For these modes the extraction region is instead taken to be the full size of the input 2D subarray or cutout for each source, or restricted to the region within which the world coordinate system (WCS) is defined in each cutout.

For slit-like 2D input data, source and background extractions are done using a rectangular aperture that covers one pixel in the dispersion direction and uses a height in the cross-dispersion direction that is defined by parameters in the EXTRACT1D reference file. For 3D IFU data, on the other hand, the extraction options differ depending on whether the target is a point or extended source. For a point source, the spectrum is extracted using circular aperture photometry, optionally including background subtraction using a circular annulus. For an extended source, rectangular aperture photometry is used, with the entire image being extracted, and no background subtraction, regardless of what was specified in the reference file or step arguments. For both point or extended sources, photometric measurements make use of the Astropy affiliated package photutils (https://photutils.readthedocs.io/en/latest/) to define an aperture object and perform extraction. For 3D NIRSpec fixed slit rateints data, the `extract_1d` step will be skipped as 3D input for the mode is not supported.

For most spectral modes an aperture correction will be applied to the extracted 1D spectral data (unless otherwise selected by the user), in order to put the results onto an infinite aperture scale. This is done by creating interpolation functions based on the APCORR reference file data and applying the interpolated aperture correction (a multiplicative factor between 0 and 1) to the extracted, 1D spectral data (corrected data include the "flux", "surf_bright", "flux_error", "sb_error", and all flux and surface brightness variance columns in the output table).

Input

Calibrated and potentially resampled 2D images or 3D cubes. The format should be a CubeModel, SlitModel, IFUCubeModel, ImageModel, MultiSlitModel, or a ModelContainer. For some JWST modes this is usually a resampled product, such as the "s2d" products for MIRI LRS fixed-slit, NIRSpec fixed-slit, and NIRSpec MOS, or the "s3d" products for MIRI MRS and NIRSpec IFU. For other modes that are not resampled (e.g. MIRI LRS slitless, NIRISS SOSS, NIRSpec BrightObj, and NIRCam and NIRISS WFSS), this will be a "cal" product. For modes that have multiple slit instances (NIRSpec fixed-slit and MOS, WFSS), the SCI extensions should have the keyword SLTNAME to specify which slit was extracted, though if there is only one slit (e.g. MIRI LRS and NIRISS SOSS), the slit name can be taken from the EXTRACT1D reference file instead.

Normally the *photom* step should be applied before running `extract_1d`. If `photom` has not been run, a warning will be logged and the output of `extract_1d` will be in units of count rate. The `photom` step converts data to units of either surface brightness (MegaJanskys per steradian) or, for point sources observed with NIRSpec and NIRISS SOSS, units of flux density (MegaJanskys).

Output

The output will be in `MultiSpecModel` format. For each input slit there will be an output table extension with the name EXTRACT1D. This extension will have columns WAVELENGTH, FLUX, FLUX_ERROR, FLUX_VAR_POISSON, FLUX_VAR_RNOISE, FLUX_VAR_FLAT, SURF_BRIGHT, SB_ERROR, SB_VAR_POISSON, SB_VAR_RNOISE, SB_VAR_FLAT, DQ, BACKGROUND, BKGD_ERROR, BKGD_VAR_POISSON, BKGD_VAR_RNOISE, BKGD_VAR_FLAT and NPIXELS. Some meta data will be written to the table header, mostly copied from the input header.

The output WAVELENGTH data is copied from the wavelength array of the input 2D data, if that attribute exists and was populated, otherwise it is calculated from the WCS. FLUX is the flux density in Janskys; see keyword TUNIT2 if the data are in a FITS BINTABLE. FLUX_ERROR is the error estimate for FLUX, and it has the same units as FLUX. The error is calculated as the square root of the sum of the three variance arrays: Poisson, read noise (RNOISE), and flat field (FLAT). SURF_BRIGHT is the surface brightness in MJy / sr, except that for point sources observed with NIRSpec and NIRISS SOSS, SURF_BRIGHT will be set to zero, because there's no way to express the extracted results from those modes as a surface brightness. SB_ERROR is the error estimate for SURF_BRIGHT, calculated in the same fashion as FLUX_ERROR but using the SB_VAR arrays. While it's expected that a user will make use of the FLUX column for point-source data and the SURF_BRIGHT column for an extended source, both columns are populated (except for NIRSpec and NIRISS SOSS point sources, as mentioned above). The `extract_1d` step collapses the input data from 2-D to 1-D by summing one or more rows (or columns, depending on the dispersion direction). A background may optionally be subtracted, but there are also other options for background subtraction prior to `extract_1d`. For the case of input data in units of MJy / sr, the SURF_BRIGHT and BACKGROUND columns are populated by dividing the sum by the number of pixels (see the NPIXELS column, described below) that were added together. The FLUX column is populated by multiplying the sum by the solid angle of a pixel, and also multiplying by 10^6 to convert from MJy to Jy. For the case of input data in units of MJy (i.e. point sources, NIRSpec or NIRISS SOSS), the SURF_BRIGHT column is set to zero, the FLUX column is just multiplied by 10^6, and the BACKGROUND column is divided by NPIXELS and by the solid angle of a pixel to convert to surface brightness (MJy / sr).

NPIXELS is the number of pixels that were added together for the source extraction region. Note that this is not necessarily a constant, and the value is not necessarily an integer (the data type is float). BACKGROUND is the measured background, scaled to the extraction width used for FLUX and SURF_BRIGHT. BACKGROUND will be zero if background subtraction is not requested. BKGD_ERROR is calculated as the square root of the sum of the BKGD_VAR arrays. DQ is not populated with useful values yet.

Extraction for 2D Slit Data

The operational details of the 1D extraction depend heavily on the parameter values given in the *EXTRACT1D* reference file. Here we describe their use within the `extract_1d` step.

Source Extraction Region

As described in the documentation for the *EXTRACT1D* reference file, the characteristics of the source extraction region can be specified in one of two different ways. The simplest approach is to use the `xstart`, `xstop`, `ystart`, `ystop`, and `extract_width` parameters. Note that all of these values are zero-indexed integers, the start and stop limits are inclusive, and the values are in the frame of the image being operated on (which could be a cutout of a larger original image). If `dispaxis=1`, the limits in the dispersion direction are `xstart` and `xstop` and the limits in the cross-dispersion direction are `ystart` and `ystop`. If `dispaxis=2`, the rolls are reversed.

If `extract_width` is also given, that takes priority over `ystart` and `ystop` (for `dispaxis=1`) for the extraction width, but `ystart` and `ystop` will still be used to define the centering of the extraction region in the cross-dispersion direction. For point source data, then the `xstart` and `xstop` values (dispaxis = 2) are shifted to account for the expected location of the source. If dispaxis=1, then the `ystart` and `ystop` values are modified. The offset amount is calculated internally. If it is not desired to apply this offset, then set `use_source_posn` = False. If the `use_source_posn` parameter is None (default), the values of `xstart/xstop` or `ystart/ystop` in the `extract_1d` reference file will be used to determine the center position of the extraction aperture. If these values are not set in the reference file, the `use_source_posn` will be set internally to True for point source data according to the table given in *srctype*. Any of the extraction location parameters will be modified internally by the step code if the extraction region would extend outside the limits of the input image or outside the domain specified by the WCS.

A more flexible way to specify the source extraction region is via the `src_coeff` parameter. `src_coeff` is specified as a list of lists of floating-point polynomial coefficients that define the lower and upper limits of the source extraction region as a function of dispersion. This allows, for example, following a tilted or curved spectral trace or simply following the variation in cross-dispersion FWHM as a function of wavelength. If both `src_coeff` and `ystart/ystop` values are given, `src_coeff` takes precedence. The `xstart` and `xstop` values can still be used to limit the range of the extraction in the dispersion direction. More details on the specification and use of polynomial coefficients is given below.

Background Extraction Regions

One or more background extraction regions for a given aperture instance can be specified using the `bkg_coeff` parameter in the EXTRACT1D reference file. This is directly analogous to the use of `src_coeff` for specifying source extraction regions and functions in exactly the same way. More details on the use of polynomial coefficients is given in the next section. Background subtraction will be done if and only if `bkg_coeff` is given in the EXTRACT1D reference file. The background is determined independently for each column (or row, if dispersion is vertical), using pixel values from all background regions within each column (or row).

Parameters related to background subtraction are `smoothing_length`, `bkg_fit`, and `bkg_order`:

1. If `smoothing_length` is specified, the 2D image data used to perform background extraction will be smoothed along the dispersion direction using a boxcar of width `smoothing_length` (in pixels). If not specified, no smoothing of the input 2D image data is performed.
2. `bkg_fit` specifies the type of background computation to be performed within each column (or row). The default value is None; if not set by the user, the step will search the reference file for a value. If no value is found, `bkg_fit` will be set to "poly". The "poly" mode fits a polynomial of order `bkg_order` to the background values within the column (or row). Alternatively, values of "mean" or "median" can be specified in order to compute the simple mean or median of the background values in each column (or row). Note that using "bkg_fit=mean" is mathematically equivalent to "bkg_fit=poly" with "bkg_order=0". If `bkg_fit` is provided both by a reference file

and by the user, e.g. `steps.extract_1d.bkg_fit='poly'`, the user-supplied value will override the reference file value.

3. If `bkg_fit=poly` is specified, `bkg_order` is used to indicate the polynomial order to be used. The default value is zero, i.e. a constant.

During source extraction, the background fit is evaluated at each pixel within the source extraction region for that column (row), and the fitted values will be subtracted (pixel by pixel) from the source count rate.

Source and Background Coefficient Lists

The interpretation and use of polynomial coefficients to specify source and background extraction regions via `src_coeff` and `bkg_coeff` is the same. The coefficients are specified as a list of an even number of lists (an even number because both the lower and upper limits of each extraction region must be specified). The source extraction coefficients will normally be a list of just two lists, the coefficients for the lower limit function and the coefficients for the upper limit function of one extraction region. The limits could just be constant values, e.g. [[324.5], [335.5]]. Straight but tilted lines are linear functions:

[[324.5, 0.0137], [335.5, 0.0137]]

Multiple regions may be specified for either the source or background, or both. It will be common to specify more than one background region. Here is an example for specifying two background regions:

[[315.2, 0.0135], [320.7, 0.0135], [341.1, 0.0139], [346.8, 0.0139]]

This is interpreted as follows:

- [315.2, 0.0135]: lower limit for first background region
- [320.7, 0.0135]: upper limit for first background region
- [341.1, 0.0139]: lower limit for second background region
- [346.8, 0.0139]: upper limit for second background region

Note: If the dispersion direction is vertical, replace "lower" with "left" and "upper" with "right" in the above description.

Notice especially that `src_coeff` and `bkg_coeff` contain floating-point values. For interpreting fractions of a pixel, the convention used here is that the pixel number at the center of a pixel is a whole number. Thus, if a lower or upper limit is a whole number, that limit splits the pixel in two, so the weight for that pixel will be 0.5. To include all the pixels between 325 and 335 inclusive, for example, the lower and upper limits would be given as 324.5 and 335.5 respectively.

The order of a polynomial is specified implicitly to be one less than the number of coefficients. The number of coefficients for a lower or upper extraction region limit must be at least one (i.e. zeroth-order polynomial). There is no predefined upper limit on the number of coefficients (and hence polynomial order). The various polynomials (lower limits, upper limits, possibly multiple regions) do not need to have the same number of coefficients; each of the inner lists specifies a separate polynomial. However, the independent variable (wavelength or pixel) does need to be the same for all polynomials for a given slit.

Polynomials specified via `src_coeff` and `bkg_coeff` are functions of either wavelength (in microns) or pixel number (pixels in the dispersion direction, with respect to the input 2D slit image), which is specified by the parameter `independent_var`. The default is "pixel". The values of these polynomial functions are pixel numbers in the direction perpendicular to dispersion.

Extraction for 3D IFU Data

In IFU cube data, 1D extraction is controlled by a different set of EXTRACT1D reference file parameters. For point source data the extraction aperture is centered at the RA/DEC target location indicated by the header. If the target location is undefined in the header, then the extraction region is the center of the IFU cube. For extended source data, anything specified in the reference file or step arguments will be ignored; the entire image will be extracted, and no background subtraction will be done.

For point sources a circular extraction aperture is used, along with an optional circular annulus for background extraction and subtraction. The size of the extraction region and the background annulus size varies with wavelength. The extraction related vectors are found in the asdf extract1d reference file. For each element in the `wavelength` vector there are three size components: `radius`, `inner_bkg`, and `outer_bkg`. The radius vector sets the extraction size; while `inner_bkg` and `outer_bkg` specify the limits of an annular background aperture. There are two additional vectors in the reference file, `axis_ratio` and `axis_pa`, which are placeholders for possible future functionality. The extraction size parameters are given in units of arcseconds and converted to units of pixels in the extraction process.

The region of overlap between an aperture and a pixel can be calculated by one of three different methods, specified by the `method` parameter: "exact" (default), limited only by finite precision arithmetic; "center", the full value in a pixel will be included if its center is within the aperture; or "subsample", which means pixels will be subsampled N x N and the "center" option will be used for each sub-pixel. When `method` is "subsample", the parameter `subpixels` is used to set the resampling value. The default value is 10.

For IFU cubes the error information is contained entirely in the ERR array, and is not broken out into the VAR_POISSON, VAR_RNOISE, and VAR_FLAT arrays. As such, `extract_1d` only propagates this non-differentiated error term. Note that while covariance is also extremely important for IFU data cubes (as the IFUs themselves are significantly undersampled) this term is not presently computed or taken into account in the `extract_1d` step. As such, the error estimates should be taken as a rough approximation that will be characterized and improved as flight data become available.

Step Arguments

The `extract_1d` step has the following step-specific arguments.

`--smoothing_length`
: If `smoothing_length` is greater than 1 (and is an odd integer), the image data used to perform background extraction will be smoothed in the dispersion direction with a boxcar of this width. If `smoothing_length` is None (the default), the step will attempt to read the value from the *EXTRACT1D* reference file. If a value is specified in the reference file, that value will be used. Note that by specifying this parameter in the *EXTRACT1D* reference file, a different value can be designated for each slit, if desired. If no value is specified either by the user or in the *EXTRACT1D* reference file, no background smoothing is done.

`--bkg_fit`
: The type of fit to perform to the background data in each image column (or row, if the dispersion is vertical). There are four allowed values: "poly", "mean", and "median", and None (the default value). If left as None, the step will search the reference file for a value - if none is found, `bkg_fit` will be set to "poly". If set to "poly", the background values for each pixel within all background regions in a given column (or row) will be fit with a polynomial of order "bkg_order" (see below). Values of "mean" and "median" compute the simple average and median, respectively, of the background region values in each column (or row). This parameter can also be specified in the *EXTRACT1D* reference file. If specified in the reference file and given as an argument to the step by the user, the user-supplied value takes precedence.

`--bkg_order`
: The order of a polynomial function to be fit to the background regions. The fit is done independently for each column (or row, if the dispersion is vertical) of the input image, and the fitted curve will be subtracted from the target data. `bkg_order` = 0 (the minimum allowed value) means to fit a constant. The user-supplied value (if any) overrides the value in the *EXTRACT1D* reference file. If neither is specified, a value of 0 will be used. If a

sufficient number of valid data points is unavailable to construct the polynomial fit, the fit will be forced to 0 for that particular column (or row). If "bkg_fit" is not "poly", this parameter will be ignored.

--bkg_sigma_clip
The background values will be sigma-clipped to remove outlier values from the determination of the background. The default value is a 3.0 sigma clip.

--log_increment
Most log messages are suppressed while looping over integrations, i.e. when the input is a CubeModel or a 3-D SlitModel. Messages will be logged while processing the first integration, but since they would be the same for every integration, most messages will only be written once. However, since there can be hundreds or thousands of integrations, which can take a long time to process, it would be useful to log a message every now and then to let the user know that the step is still running.

`log_increment` is an integer, with default value 50. If it is greater than 0, an INFO message will be printed every `log_increment` integrations, e.g. "... 150 integrations done".

--subtract_background
This is a boolean flag to specify whether the background should be subtracted. If None, the value in the *EXTRACT1D* reference file (if any) will be used. If not None, this parameter overrides the value in the reference file.

--use_source_posn
This is a boolean flag to specify whether the target and background extraction region locations specified in the *EXTRACT1D* reference file should be shifted to account for the expected position of the source. If None (the default), the step will make the decision of whether to use the source position based on the observing mode and the source type. The source position will only be used for point sources and for modes where the source could be located off-center due to things like nodding or dithering. If turned on, the position of the source is used in conjunction with the World Coordinate System (WCS) to compute the x/y source location. For NIRSpec non-IFU modes, the source position is determined from the `source_xpos/source_ypos` parameters. For MIRI LRS fixed slit, the dither offset is applied to the sky pointing location to determine source position. All other modes use `targ_ra/targ_dec`. If this parameter is specified in the *EXTRACT1D* reference file, the reference file value will override any automatic settings based on exposure and source type. As always, a value given by the user as an argument to the step overrides all settings in the reference file or within the step code.

--center_xy
A list of two integer values giving the desired x/y location for the center of the circular extraction aperture used for extracting spectra from 3-D IFU cubes. Ignored for non-IFU modes and non-point sources. Must be given in x,y order and in units of pixels along the x,y axes of the 3-D IFU cube, e.g. `--center_xy="27,28"`. If given, the values override any position derived from the use of the `use_source_posn` argument. Default is None.

--apply_apcorr
Switch to select whether or not to apply an APERTURE correction during the Extract1dStep processing. Default is `True`

--ifu_autocen
Switch to select whether or not to enable auto-centroiding of the extraction aperture for IFU point sources. Auto-centroiding works by median collapsing the IFU cube across all wavelengths and using DAOStarFinder to locate the brightest source in the field. Default is `False`.

--ifu_rfcorr
Switch to select whether or not to run 1d residual fringe correction on the extracted 1d spectrum (MIRI MRS only). Default is `False`.

--ifu_set_srctype
A string that can be used to override the extraction method for the source_type given by the SRCTYPE keyword. The allowed values are POINT and EXTENDED. The SRCTYPE keyword is not changed, instead the extraction method used is based on this parameter setting. This is only allowed for MIRI MRS IFU data.

--ifu_rscale
A float designating the number of PSF FWHMs to use for the extraction radius. This is a MIRI MRS only paramenter. Values accepted are between 0.5 to 3.0. The default extraction size is set to 2 * FWHM. Values below 2 will result in a smaller radius, a value of 2 results in no change to radius and a value above 2 results in a larger extraction radius.

--soss_atoca
This is a NIRISS-SOSS algorithm-specific parameter; if True, use the ATOCA algorithm to treat order contamination. Default is `True`.

--soss_threshold
This is a NIRISS-SOSS algorithm-specific parameter; this sets the threshold value for a pixel to be included when modelling the spectral trace. The default value is 0.01.

--soss_n_os
This is a NIRISS-SOSS algorithm-specific parameter; this is an integer that sets the oversampling factor of the underlying wavelength grid used when modeling the trace. The default value is 2.

--soss_estimate
This is a NIRISS-SOSS algorithm-specific parameter; filename or SpecModel of the estimate of the target flux. The estimate must be a SpecModel with wavelength and flux values.

--soss_wave_grid_in
This is a NIRISS-SOSS algorithm-specific parameter; filename or SossWaveGridModel containing the wavelength grid used by ATOCA to model each valid pixel of the detector. If not given, the grid is determined based on an estimate of the flux (soss_estimate), the relative tolerance (soss_rtol) required on each pixel model and the maximum grid size (soss_max_grid_size).

--soss_wave_grid_out
This is a NIRISS-SOSS algorithm-specific parameter; filename to hold the wavelength grid calculated by ATOCA, stored in a SossWaveGridModel.

--soss_rtol
This is a NIRISS-SOSS algorithm-specific parameter; the relative tolerance needed on a pixel model. It is used to determine the sampling of the soss_wave_grid when not directly given. Default value is 1.e-4.

--soss_max_grid_size
This is a NIRISS-SOSS algorithm-specific parameter; the maximum grid size allowed. It is used when soss_wave_grid is not provided to make sure the computation time or the memory used stays reasonable. Default value is 20000.

--soss_transform
This is a NIRISS-SOSS algorithm-specific parameter; this defines a rotation to apply to the reference files to match the observation. It should be specified as a list of three floats, with default values of None.

--soss_tikfac
This is a NIRISS-SOSS algorithm-specific parameter; this is the regularization factor used in the SOSS extraction. If not specified, ATOCA will calculate a best-fit value for the Tikhonov factor.

--soss_width
This is a NIRISS-SOSS algorithm-specific parameter; this specifies the aperture width used to extract the 1D spectrum from the decontaminated trace. The default value is 40.0 pixels.

--soss_bad_pix
This is a NIRISS-SOSS algorithm-specific parameter; this parameter sets the method used to handle bad pixels. There are currently two options: "model" will replace the bad pixel values with a modeled value, while "masking" will omit those pixels from the spectrum. The default value is "model".

--soss_modelname
This is a NIRISS-SOSS algorithm-specific parameter; if set, this will provide the optional ATOCA model output

of traces and pixel weights, with the filename set by this parameter. By default this is set to None and this output is not provided.

Reference File

The `extract_1d` step uses an EXTRACT1D reference file and an APCORR reference file.

EXTRACT1D Reference File

The EXTRACT1D reference file contains information needed to guide the 1D extraction process. It is a text file, with the information in JSON format for Non-IFU data and in ASDF format for IFU data.

Reference Selection Keywords for EXTRACT1D

CRDS selects appropriate EXTRACT1D references based on the following keywords. EXTRACT1D is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, LAMP, OPMODE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for EXTRACT1D

In addition to the standard reference file keywords listed above, the following keywords are *required* in EXTRACT1D reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for EXTRACT1D*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

Reference File Format for non-IFU data

EXTRACT1D reference files for non-IFU data are text files, with the information stored in JSON format. All the information is specified in a list with key `apertures`. Each element of this list is a dictionary, one for each aperture (e.g. a slit) that is supported by the given reference file. The particular dictionary used by the step is found by matching the slit name in the science data with the value of key `id`. Key `spectral_order` is optional, but if it is present, it must match the expected spectral order number.

The following keys are supported for non-IFU data (see below for IFU keys). Key `id` is the primary criterion for selecting which element of the `apertures` list to use. The slit name (except for a full-frame input image) is compared with the values of `id` in the `apertures` list to select the appropriate aperture. In order to allow the possibility of multiple spectral orders for the same slit name, there may be more than one element of `apertures` with the same value for key `id`. These should then be distinguished by using the secondary selection criterion `spectral_order`. In this case, the various spectral orders would likely have different extraction locations within the image, so different elements of `apertures` are needed in order to specify those locations. If key `dispaxis` is specified, its value will be used to set the dispersion direction within the image. If `dispaxis` is not specified, the dispersion direction will be taken to be the axis along which the wavelengths change more rapidly. Key `region_type` can be omitted, but if it is specified, its value must be "target". The remaining keys specify the characteristics of the source and background extraction regions.

- id: the slit name, e.g. "S200A1" (string)
- spectral_order: the spectral order number (optional); this can be either positive or negative, but it should not be zero (int)
- dispaxis: dispersion direction, 1 for X, 2 for Y (int)
- xstart: first pixel in the horizontal direction, X (int) (0-indexed)
- xstop: last pixel in the horizontal direction, X (int) (0-indexed)
- ystart: first pixel in the vertical direction, Y (int) (0-indexed)
- ystop: last pixel in the vertical direction, Y (int) (0-indexed)
- src_coeff: this takes priority for specifying the source extraction region (list of lists of float)
- bkg_coeff: for specifying background subtraction regions (list of lists of float)
- independent_var: "wavelength" or "pixel" (string)
- smoothing_length: width of boxcar for smoothing background regions along the dispersion direction (odd int)
- bkg_fit: the type of background fit or computation (string)
- bkg_order: order of polynomial fit to background regions (int)
- extract_width: number of pixels in cross-dispersion direction (int)
- use_source_posn: adjust the extraction limits based on source RA/Dec (bool)

Note: All parameter values that represent pixel indexes, such as `xstart`, `xstop`, and the `src_coeff` and `bkg_coeff` coefficients, are always in the frame of the image being operated on, which could be a small cutout from a larger original image. They are also ZERO-indexed and the limits are inclusive (e.g. 11-15 includes 5 pixels).

See *Extraction for 2D Slit Data* for more details on how these parameters are used in the extraction process.

Editing JSON Reference File Format for non-IFU data

The default EXTRACT1D reference file is found in CRDS. The user can download this file, modify the contents, and use this modified file in `extract_1d` by specifying this modified reference file with the `override` option (*override in python* or *override in strun*). The format for JSON files has to be exact, for example, the format of a floating-point value with a fractional portion must include at least one decimal digit, so "1." is invalid, while "1.0" is valid. The best practice after editing a JSON reference file is to run a JSON validator off-line, such as `https://jsonlint.com/`, and correct any format errors before using the JSON reference file in the pipeline.

Reference File Format IFU data

For IFU data the reference files are stored as ASDF files. The extraction size varies with wavelength. The reference file contains a set of vectors defining the extraction size based on wavelength. These vectors are all the same size and are defined as follows:

- wavelength: wavelength in microns.
- radius: the radius of the circular extraction aperture (arcseconds, float)
- inner_bkg: of the inner edge of the background annulus (arcseconds, float)
- outer_bkg: of the outer edge of the background annulus (arcseconds, float)

In addition following general keys are also in the ASDF reference file:

- subtract_background: if true, subtract a background determined from an annulus with inner and outer radii given by `inner_bkg` and `outer_bkg` (boolean)
- method: one of "exact", "subpixel", or "center", the method used by photutils for computing the overlap between apertures and pixels (string, default is "exact")
- subpixels: if `method` is "subpixel", pixels will be resampled by this factor in each dimension (int, the default is 10)

See *Extraction for 3D IFU Data* for more details on how these parameters are used in the extraction process.

Example EXTRACT1D Reference File

The following JSON was taken as an example from reference file jwst_niriss_extract1d_0003.json:

```
{
    "REFTYPE": "EXTRACT1D",
    "INSTRUME": "NIRISS",
    "TELESCOP": "JWST",
    "DETECTOR": "NIS",
    "EXP_TYPE": "NIS_SOSS",
    "PEDIGREE": "GROUND",
```

(continues on next page)

(continued from previous page)


```
    "DESCRIP": "NIRISS SOSS extraction params for ground testing",
    "AUTHOR": "M.Wolfe, H.Bushouse",
    "HISTORY": "Build 7.1 of the JWST Calibration pipeline. The regions are rectangular␣
↪and do not follow the trace.",
    "USEAFTER": "2015-11-01T00:00:00",
    "apertures": [
      {
      "id": "FULL",
      "region_type": "target",
      "bkg_coeff": [[2014.5],[2043.5]],
      "xstart":    4,
      "xstop":  2044,
      "ystart": 1792,
      "ystop":  1972,
      "dispaxis": 1,
      "extract_width": 181
      },

      {
      "id": "SUBSTRIP256",
      "region_type": "target",
      "bkg_coeff": [[221.5],[251.5]],
      "xstart":   4,
      "xstop": 2044,
      "ystart": 20,
      "ystop":  220,
      "dispaxis": 1,
      "extract_width": 201
      },

      {
      "id": "SUBSTRIP96",
      "region_type": "target",
      "bkg_coeff": [[1.5],[8.5],[92.5],[94.5]],
      "xstart":   4,
      "xstop": 2044,
      "ystart":  10,
      "ystop":   92,
      "dispaxis": 1,
      "extract_width": 83
      }]
}
```

APCORR Reference File

REFTYPE
APCORR

The APCORR reference file contains data necessary for correcting extracted imaging and spectroscopic photometry to the equivalent of an infinite aperture. It is used within the *source_catalog* step for imaging and within the *extract_1d* step for spectroscopic data.

Reference Selection Keywords for APCORR

CRDS selects appropriate APCORR references based on the following keywords. APCORR is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, FILTER, GRATING, LAMP, OPMODE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for APCORR

In addition to the standard reference file keywords listed above, the following keywords are *required* in APCORR reference files, because they are used as CRDS selectors (see apcorr_selectors):

Keyword	Data Model Name	Instruments
EXP_TYPE	model.meta.exposure.type	All

NON-IFU APCORR Reference File Format

APCORR reference files for non-IFU data are in FITS format. The FITS APCORR reference file contains tabular data in a BINTABLE extension with EXTNAME = 'APCORR'. The FITS primary HDU does not contain a data array. The contents of the table extension varies for different instrument modes, as shown in the tables below.

Data model

`FgsImgApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.FgsImgApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
FGS	Image	eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

`MirImgApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirImgApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	Image	filter	string	12	N/A
		subarray	string	15	N/A
		eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

`MirLrsApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirLrsApcorrModel.html#jwst.dat

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	LRS	subarray	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	1D array	pixels
		nelem_size	integer	scalar	N/A
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model

NrcImgApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrcImgApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRCam	Image	filter	string	12	N/A
		pupil	string	15	N/A
		eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

NrcWfssApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrcWfssApcorrModel.html#jwst.

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRCam	WFSS	filter	string	12	N/A
		pupil	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	1D array	pixels
		nelem_size	integer	scalar	N/A
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model

NisImgApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NisImgApcorrModel.html#jwst.dat

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRISS	Image	filter	string	12	N/A
		pupil	string	15	N/A
		eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

NisWfssApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NisWfssApcorrModel.html#jwst.

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRISS	WFSS	filter	string	12	N/A
		pupil	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	1D array	pixels
		nelem_size	integer	scalar	N/A
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model

NrsFsApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrsFsApcorrModel.html#jwst.datam

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	FS	filter	string	12	N/A
		grating	string	15	N/A
		slit	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	2D array	arcsec
		nelem_size	integer	scalar	N/A
		pixphase	float	1D array	N/A
		apcorr	float	3D array	unitless
		apcorr_err	float	3D array	unitless

Data model

NrsMosApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrsMosApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	MOS	filter	string	12	N/A
		grating	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	2D array	arcsec
		nelem_size	integer	scalar	N/A
		pixphase	float	1D array	N/A
		apcorr	float	3D array	unitless
		apcorr_err	float	3D array	unitless

Row Selection

A row of data within the reference table is selected by the pipeline step based on the optical elements in use for the exposure. The selection attributes are always contained in the first few columns of the table. The remaining columns contain the data needed for the aperture correction. The row selection criteria for each instrument+mode are:

- **FGS Image:**
  - None (table contains a single row)
- **MIRI:**
  - Image: Filter and Subarray
  - LRS: Subarray
- **NIRCam:**
  - Image: Filter and Pupil
  - WFSS: Filter and Pupil
- **NIRISS:**
  - Image: Filter and Pupil
  - WFSS: Filter and Pupil

•**NIRSpec:**

- MOS: Filter and Grating
- Fixed Slits: Filter, Grating, and Slit name

Note: Spectroscopic mode reference files contain the "nelem_wl" and "nelem_size" entries, which indicate to the pipeline step how many valid elements are contained in the "wavelength" and "size" arrays, respectively. Only the first "nelem_wl" and "nelem_size" entries are read from each array.

IFU APCORR Reference File ASDF Format

For IFU data the APCORR reference files are in ASDF format. The aperture correction varies with wavelength and the contents of the files are shown below. The radius, aperture correction and error are all 2D arrays. Currently the 2nd dimension does not add information, but in the future it could be used to provide different aperture corrections for different radii.

Data model

`MirMrsApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirMrsApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	MRS	wavelength	float	1D array	micron
		radius	float	2D array	arcsec
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model

`NRSIFUApcorrModel`

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	MOS	filter	string	12	N/A
		grating	string	15	N/A
		wavelength	float	1D array	micron
		radius	float	2D array	arcsec
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Reference Image Format

An alternative EXTRACT1D reference format, an image, is also supported. There are currently no files of this type in CRDS (there would be a conflict with the current JSON-format reference files), but a user can create a file in this format and specify that it be used as an override for the default EXTRACT1D reference file.

This format is a `MultiExtract1dImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiExtract1dImageM which is loosely based on `MultiSlitModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jws The file should contain keyword DATAMODL, with value 'MultiExtract1dImageModel'; this is not required, but it makes it possible to open the file simply with `datamodels.open`. The reference image file contains one or more images, which are of type `Extract1dImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.Extract1dImageModel.html#jwst.datamodels.Extract1dImageModel), and one can iterate over the list of these images to find one that matches the observing configuration. This iterable is the `images` attribute of the model (`ref_model`, for purposes of discussion). Each element of `ref_model.images` can contain a `name` attribute (FITS keyword SLTNAME) and a `spectral_order` attribute (FITS keyword SPORDER),

which can be compared with the slit name and spectral order respectively in the science data model in order to select the matching reference image. The wildcard for SLTNAME is "ANY", and any integer value for SPORDER greater than or equal to 1000 is a wildcard for spectral order (SPORDER is an integer, and an integer keyword may not be assigned a string value such as "ANY"). For IFU data, the image to use is selected only on `name`.

For non-IFU data, the shape of the reference image should match the shape of the science data, although the step can either trim the reference image or pad it with zeros to match the size of the science data, pinned at pixel [0, 0]. For IFU data, the shape of the reference image can be 3-D, exactly matching the shape of the IFU data, or it can be 2-D, matching the shape of one plane of the IFU data. If the reference image is 2-D, it will be applied equally to each plane of the IFU data, i.e. it will be broadcast over the dispersion direction.

The data type of each image is float32, but the data values may only be +1, 0, or -1. A value of +1 means that the matching pixel in the science data will be included when accumulating data for the source (target) region. A value of 0 means the pixel will not be used for anything. A value of -1 means the pixel will be included for the background; if there are no pixels with value -1, no background will be subtracted. A pixel will either be included or not; there is no option to include only a fraction of a pixel.

For non-IFU data, values will be extracted column by column (if the dispersion direction is horizontal, else row by row). The gross count rate will be the sum of the source pixels in a column (or row). If background region(s) were specified, the sum of those pixels will be scaled by the ratio of the number of source pixels to the number of background pixels (with possibly a different ratio for each column (row)) before being subtracted from the gross count rate. The scaled background is what will be saved in the output table.

For IFU data, the values will be summed over each plane in the dispersion direction, giving one value of flux and optionally one value of background per plane. The background value will be scaled by the ratio of source pixels to background pixels before being subtracted from the flux.

jwst.extract_1d Package

Classes

Extract1dStep([name, parent, config_file, ...])	Extract a 1-d spectrum from 2-d data

Extract1dStep

class jwst.extract_1d.**Extract1dStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Extract a 1-d spectrum from 2-d data

smoothing_length

If not None, the background regions (if any) will be smoothed with a boxcar function of this width along the dispersion direction. This should be an odd integer.

Type

int (https://docs.python.org/3/library/functions.html#int) or None

bkg_fit

A string indicating the type of fitting to be applied to background values in each column (or row, if the dispersion is vertical). Allowed values are `poly`, `mean`, and `median`. Default is `None` (https://docs.python.org/3/library/constants.html#None).

Type
str (https://docs.python.org/3/library/stdtypes.html#str)

bkg_order

If not None, a polynomial with order *bkg_order* will be fit to each column (or row, if the dispersion direction is vertical) of the background region or regions. For a given column (row), one polynomial will be fit to all background regions. The polynomial will be evaluated at each pixel of the source extraction region(s) along the column (row), and the fitted value will be subtracted from the data value at that pixel. If both *smoothing_length* and *bkg_order* are not None, the boxcar smoothing will be done first.

Type
int (https://docs.python.org/3/library/functions.html#int) or None

bkg_sigma_clip

Background sigma clipping value to use on background to remove outliers and maximize the quality of the 1d spectrum

Type
float (https://docs.python.org/3/library/functions.html#float)

log_increment

if *log_increment* is greater than 0 (the default is 50) and the input data are multi-integration (which can be CubeModel or SlitModel), a message will be written to the log with log level INFO every *log_increment* integrations. This is intended to provide progress information when invoking the step interactively.

Type
int (https://docs.python.org/3/library/functions.html#int)

subtract_background

A flag which indicates whether the background should be subtracted. If None, the value in the extract_1d reference file will be used. If not None, this parameter overrides the value in the extract_1d reference file.

Type
bool (https://docs.python.org/3/library/functions.html#bool) or None

use_source_posn

If True, the source and background extraction positions specified in the extract1d reference file (or the default position, if there is no reference file) will be shifted to account for the computed position of the source in the data. If None (the default), the values in the reference file will be used. Aperture offset is determined by computing the pixel location of the source based on its RA and Dec. It does not make sense to apply aperture offsets for extended sources, so this parameter can be overridden (set to False) internally by the step.

Type
bool (https://docs.python.org/3/library/functions.html#bool) or None

center_xy

A list of 2 pixel coordinate values at which to place the center of the IFU extraction aperture, overriding any centering done by the step. Two values, in x,y order, are used for extraction from IFU cubes. Default is None.

Type
int (https://docs.python.org/3/library/functions.html#int) or None

apply_apcorr

Switch to select whether or not to apply an APERTURE correction during the Extract1dStep. Default is True

Type
bool (https://docs.python.org/3/library/functions.html#bool)

ifu_autocen

Switch to turn on auto-centering for point source spectral extraction in IFU mode. Default is False.

Type
bool (https://docs.python.org/3/library/functions.html#bool)

ifu_rfcorr

Switch to select whether or not to apply a 1d residual fringe correction for MIRI MRS IFU spectra. Default is False.

Type
bool (https://docs.python.org/3/library/functions.html#bool)

ifu_set_srctype

For MIRI MRS IFU data override srctype and set it to either POINT or EXTENDED.

Type
str (https://docs.python.org/3/library/stdtypes.html#str)

ifu_rscale

For MRS IFU data a value for changing the extraction radius. The value provided is the number of PSF FWHMs to use for the extraction radius. Values accepted are between 0.5 to 3.0. The default extraction size is set to 2 * FWHM. Values below 2 will result in a smaller radius, a value of 2 results in no change to the radius and a value above 2 results in a larger extraction radius.

Type
float (https://docs.python.org/3/library/functions.html#float)

soss_atoca

Switch to toggle extraction of SOSS data with the ATOCA algorithm. WARNING: ATOCA results not fully validated, and require the photom step be turned off. Default is False, meaning SOSS data use box extraction.

Type
bool (https://docs.python.org/3/library/functions.html#bool), default=False

soss_threshold

Threshold value above which a pixel will be included when modeling the SOSS trace in ATOCA. Default is 0.01.

Type
float (https://docs.python.org/3/library/functions.html#float)

soss_n_os

Oversampling factor of the underlying wavelength grid when modeling the SOSS trace in ATOCA. Default is 2.

Type
int (https://docs.python.org/3/library/functions.html#int)

soss_transform

Rotation applied to the reference files to match the observation orientation. Default is None.

Type
list (https://docs.python.org/3/library/stdtypes.html#list)[float (https://docs.python.org/3/library/functions.html#float)]

soss_tikfac

The regularization factor used for extraction in ATOCA. If left to default value of None, ATOCA will find an optimized value.

Type
float (https://docs.python.org/3/library/functions.html#float)

`soss_width`

Aperture width used to extract the SOSS spectrum from the decontaminated trace in ATOCA. Default is 40.

Type
float (https://docs.python.org/3/library/functions.html#float)

`soss_bad_pix`

Method used to handle bad pixels, accepts either "model" or "masking". Default method is "model".

Type
str (https://docs.python.org/3/library/stdtypes.html#str)

`soss_modelname`

Filename for optional model output of ATOCA traces and pixel weights.

Type
str (https://docs.python.org/3/library/stdtypes.html#str)

`soss_estimate`

Filename or SpecModel of the estimate of the target flux. The estimate must be a SpecModel with wavelength and flux values.

Type
str (https://docs.python.org/3/library/stdtypes.html#str) or SpecModel or None

`soss_wave_grid_in`

Filename or SossWaveGrid containing the wavelength grid used by ATOCA to model each pixel valid pixel of the detector. If not given, the grid is determined based on an estimate of the flux (soss_estimate), the relative tolerance (soss_rtol) required on each pixel model and the maximum grid size (soss_max_grid_size).

Type
str (https://docs.python.org/3/library/stdtypes.html#str) or SossWaveGrid or None

`soss_wave_grid_out`

Filename to hold the wavelength grid calculated by ATOCA.

Type
str (https://docs.python.org/3/library/stdtypes.html#str) or None

`soss_rtol`

The relative tolerance needed on a pixel model. It is used to determine the sampling of the soss_wave_grid when not directly given.

Type
float (https://docs.python.org/3/library/functions.html#float)

`soss_max_grid_size`

Maximum grid size allowed. It is used when soss_wave_grid is not provided to make sure the computation time or the memory used stays reasonable.

Type
int (https://docs.python.org/3/library/functions.html#int)

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	Execute the step.

Attributes Documentation

class_alias = 'extract_1d'

reference_file_types = ['extract1d', 'apcorr', 'wavemap', 'spectrace', 'specprofile', 'speckernel']

spec

```
smoothing_length = integer(default=None)  # background smoothing size
bkg_fit = option("poly", "mean", "median", None, default=None)  # background␣
↪fitting type
bkg_order = integer(default=None, min=0)  # order of background polynomial fit
bkg_sigma_clip = float(default=3.0)  # background sigma clipping threshold
log_increment = integer(default=50)  # increment for multi-integration log␣
↪messages
subtract_background = boolean(default=None)  # subtract background?
use_source_posn = boolean(default=None)  # use source coords to center␣
↪extractions?
center_xy = float_list(min=2, max=2, default=None)  # IFU extraction x/y center
apply_apcorr = boolean(default=True)  # apply aperture corrections?
ifu_autocen = boolean(default=False) # Auto source centering for IFU point␣
↪source data.
ifu_rfcorr = boolean(default=False) # Apply 1d residual fringe correction
```

(continues on next page)

(continued from previous page)

```
ifu_set_srctype = option("POINT", "EXTENDED", None, default=None) # user-
↪supplied source type
ifu_rscale = float(default=None, min=0.5, max=3) # Radius in terms of PSF FWHM␣
↪to scale extraction radii
soss_atoca = boolean(default=True)  # use ATOCA algorithm
soss_threshold = float(default=1e-2)  # TODO: threshold could be removed from␣
↪inputs. Its use is too specific now.
soss_n_os = integer(default=2)  # minimum oversampling factor of the underlying␣
↪wavelength grid used when modeling trace.
soss_wave_grid_in = input_file(default = None)  # Input wavelength grid used to␣
↪model the detector
soss_wave_grid_out = string(default = None)  # Output wavelength grid solution␣
↪filename
soss_estimate = input_file(default = None)  # Estimate used to generate the␣
↪wavelength grid
soss_rtol = float(default=1.0e-4)  # Relative tolerance needed on a pixel model
soss_max_grid_size = integer(default=20000)  # Maximum grid size, if wave_grid␣
↪not specified
soss_transform = list(default=None, min=3, max=3)  # rotation applied to the␣
↪ref files to match observation.
soss_tikfac = float(default=None)  # regularization factor for NIRISS SOSS␣
↪extraction
soss_width = float(default=40.)  # aperture width used to extract the 1D␣
↪spectrum from the de-contaminated trace.
soss_bad_pix = option("model", "masking", default="masking")  # method used to␣
↪handle bad pixels
soss_modelname = output_file(default = None)  # Filename for optional model␣
↪output of traces and pixel weights
```

Methods Documentation

process(*input*)

Execute the step.

Parameters

input (*JWST data model*) –

Returns

This will be `input_model` if the step was skipped; otherwise, it will be a model containing 1-D extracted spectra.

Return type

JWST data model

Class Inheritance Diagram

15.1.19 Extract 2D Spectra

Description

Class
jwst.extract_2d.Extract2dStep

Alias
extract_2d

Overview

The `extract_2d` step extracts 2D arrays from spectral images. The extractions are performed within all of the SCI, ERR, and DQ arrays of the input image model, as well as any variance arrays that may be present. It also computes an array of wavelengths to attach to the extracted data. The extracted arrays are stored as one or more `slit` objects in an output `MultiSlitModel` and saved as separate tuples of extensions in the output FITS file.

Assumptions

This step uses the `bounding_box` attribute of the WCS stored in the data model, which is populated by the `assign_wcs` step. Hence the `assign_wcs` step must be applied to the science exposure before running this step.

For NIRCam and NIRISS WFSS modes, no `bounding_box` is attached to the data model by the `assign_wcs` step. This is to keep the WCS flexible enough to be used with any source catalog or similar list of objects that may be associated with the dispersed image. Instead, there is a helper method that is used to calculate the bounding boxes that contain the spectral trace for each object. One box is made for each spectral order of each object. In regular, automated processing, the `extract_2d` step uses the source catalog specified in the input model's meta information to create the list of objects and their corresponding bounding box. This list is used to make the 2D cutouts from the dispersed image.

NIRCam TSGRISM exposures do not use a source catalog, so the step instead relies on the assumption that the source of interest is located at the aperture reference point and centers the extraction there. More details are given below.

Algorithm

This step is currently applied only to NIRSpec Fixed-Slit, NIRSpec MOS, NIRSpec TSO (BrightObj), NIRCam and NIRISS WFSS, and NIRCam TSGRISM observations.

NIRSpec Fixed Slit and MOS

If the step parameter `slit_name` is left unspecified, the default behavior is to extract all slits that project onto the detector. A single slit may be extracted by specifying the slit name with the `slit_name` argument. In the case of a NIRSpec fixed-slit exposure the allowed slit names are: "S1600A1", "S200A1", "S200A2", "S200B1" and "S400A1". For NIRSpec MOS exposures, the slit name is the slitlet number from the MSA metadata file, corresponding to the value of the "SLTNAME" keyword in FITS products, and it has to be provided as a string, e.g. slit_name='60'.

To find out what slits are available for extraction:

```
>>> from jwst.assign_wcs import nirspec
>>> nirspec.get_open_slits(input_model)
```

The corner locations of the regions to be extracted are determined from the `bounding_box` contained in the exposure's WCS, which defines the range of valid inputs along each axis. The input coordinates are in the image frame, i.e. subarray shifts are accounted for.

The output `MultiSlitModel` data model will have the meta data associated with each slit region populated with the name and region characteristic for the slits, corresponding to the FITS keywords "SLTNAME", "SLTSTRT1", "SLTSIZE1", "SLTSTRT2", and "SLTSIZE2." Keyword "DISPAXIS" (dispersion direction) will be copied from the input file to each of the output cutout images.

NIRCam and NIRISS WFSS

During normal, automated processing of WFSS grism images, the step parameter `grism_objects` is left unspecified, in which case the `extract_2d` step uses the source catalog that is specified in the input model's meta information, `input_model.meta.source_catalog.filename` ("SCATFILE" keyword) to define the list of objects to be extracted. Otherwise, a user can submit a list of `GrismObjects` that contains information about the objects that they want extracted. The `GrismObject` list can be created automatically by using the method in `jwst.assign_wcs.utils.create_grism_bbox`. This method also uses the name of the source catalog saved in the input model's meta information. If it's better to construct a list of `GrismObjects` outside of these, the `GrismObject` itself can be imported from `jwst.transforms.models`.

The dispersion direction will be documented by copying keyword "DISPAXIS" (1 = horizontal, 2 = vertical) from the input file to the output cutout.

The `wfss_mmag_extract` and `wfss_nbright` parameters both affect which objects from a source catalog will be retained for extraction. The rejection or retention of objects proceeds as follows:

1. As each object is read from the source catalog, they are immediately rejected if their isophotal_abmag > `wfss_mmag_extract`, meaning that only objects brighter than `wfss_mmag_extract` will be retained. The default `wfss_mmag_extract` value of `None` retains all objects.
2. If the computed footprint (bounding box) of the spectral trace of an object lies completely outside the field of view of the grism image, it is rejected.
3. The list of objects retained after the above two filtering steps have been applied is sorted based on `isophotal_abmag` (listed for each source in the source catalog) and only the brightest `wfss_nbright` objects are retained. The default value of `wfss_nbright` is currently 1000.

All remaining objects are then extracted from the grism image.

WFSS Examples

The extraction of sources from WFSS grism images is a multi-step process, as outlined above. Here we show detailed examples of how to customize the list of WFSS grism objects to be extracted, in order to better explain the various steps. First, the input file (or data model) must aleady have a WCS object assigned to it by running the *assign_wcs* step. The default values for the wavelength range of each spectral order to be extracted are also required; they are stored in the `wavelengthrange` reference file, which can be retrieved from CRDS.

Load the grism image, which is assumed to have already been processed through `assign_wcs`, into an `ImageModel` data model (used for all 2-D "images", regardless of whether they actually contain imaging data or dispersed spectra):

```
>>> from stdatamodels.jwst.datamodels import ImageModel
>>> input_model = ImageModel("jw12345001001_03101_00001_nis_assign_wcs.fits")
```

Load the `extract_2d` step and retrieve the `wavelengthrange` reference file specific for this mode:

```
>>> from jwst.extract_2d import Extract2dStep
>>> step = Extract2dStep()
>>> refs = {}
>>> reftype = 'wavelengthrange'
>>> refs[reftype] = step.get_reference_file(input_model, reftype)
>>> print(refs)
{'wavelengthrange': '/crds/jwst/references/jwst_niriss_wavelengthrange_0002.asdf'}
```

Create a list of grism objects for a specified spectral order with a limited minimum magnitude and a specified half-height of the extraction box in the cross-dispersion direction via the `wfss_extract_half_height` parameter. Note that the half-height parameter only applies to point sources.

```
>>> from jwst.assign_wcs.util import create_grism_bbox
>>> grism_objects = create_grism_bbox(input_model, refs, mmag_extract=17,
... extract_orders=[1], wfss_extract_half_height=10)
>>> print(len(grism_objects))
6
>>> print(grism_objects[0])
id: 432
order_bounding {1: ((array(1113), array(1471)), (array(1389), array(1609)))}
sky_centroid: <SkyCoord (ICRS): (ra, dec) in deg
    (3.59204081, -30.40553435)>
sky_bbox_ll: <SkyCoord (ICRS): (ra, dec) in deg
    (3.59375611, -30.40286617)>
sky_bbox_lr: <SkyCoord (ICRS): (ra, dec) in deg
    (3.59520565, -30.40665425)>
sky_bbox_ur: <SkyCoord (ICRS): (ra, dec) in deg
    (3.58950974, -30.4082754)>
sky_bbox_ul:<SkyCoord (ICRS): (ra, dec) in deg
    (3.5880604, -30.40448726)>
xcentroid: 1503.6541213285695
ycentroid: 1298.2882813663837
partial_order: {1: False}
waverange: {1: (0.97, 1.32)}
is_extended: True
isophotal_abmag: 16.185488680084294
```

Create a list of grism objects for a specified spectral order and wavelength range. In this case we don't use the default wavelength range limits from the `wavelengthrange` reference file, but instead designate custom limits via

the wavelength_range parameter passed to the create_grism_bbox function, which is a dictionary of the form {spectral_order: (wave_min, wave_max)}. Use the source ID, sid, to identify the object(s) to be modified. The computed extraction limits are stored in the order_bounding attribute, which is ordered (y, x).

```
>>> from jwst.assign_wcs.util import create_grism_bbox
>>> grism_objects = create_grism_bbox(input_model, mmag_extract=18,
... wavelength_range={1: (3.01, 4.26)})
>>> print([obj.sid for obj in grism_objects])
[12, 26, 31, 37, 104]
>>> print(grism_objects[-1])
id: 104
order_bounding {1: ((array(1165), array(1566)), (array(1458), array(1577)))}
sky_centroid: <SkyCoord (ICRS): (ra, dec) in deg
    (3.57958792, -30.40926139)>
sky_bbox_ll: <SkyCoord (ICRS): (ra, dec) in deg
    (3.58060118, -30.40800999)>
sky_bbox_lr: <SkyCoord (ICRS): (ra, dec) in deg
    (3.58136873, -30.41001654)>
sky_bbox_ur: <SkyCoord (ICRS): (ra, dec) in deg
    (3.57866098, -30.4107869)>
sky_bbox_ul:<SkyCoord (ICRS): (ra, dec) in deg
    (3.57789348, -30.40878033)>
xcentroid: 1513.4964315117466
ycentroid: 1920.6251490007467
partial_order: {1: False}
waverange: {1: (3.01, 4.26)}
is_extended: True
isophotal_abmag: 17.88278103874113
>>> grism_object[-1].order_bounding[1] = ((1200, 1500), (1480, 1520))
>>> print(grism_object[-1].order_bounding
{1: ((1200, 1500), (1480,1520))}
```

The grism_objects list created in the above examples can now be used as input to the extract_2d step in order to extract the particular objects defined in that list:

```
>>> result = step.call(input_model, grism_objects=grism_objects)
```

result is a MultiSlitModel data model, containing one SlitModel instance for each extracted object, which includes meta data that identify each object and the actual extracted data arrays, e.g.:

```
>>> print(len(result.slits))
8
>>> result.slits[0].source_id
104
>>> result.slits[0].data
array([[..., ..., ...]])
```

NIRCam TSGRISM

There is no source catalog created for TSO grism observations, because no associated direct images are obtained from which to derive such a catalog. So the `extract_2d` step relies on the fact that the source of interest is placed at the aperture reference point to determine the source location. The aperture reference location, in units of image x and y pixels, is read from the keywords "XREF_SCI" and "YREF_SCI" in the SCI extension header of the input image. These values are used to set the source location for all computations involving the extent of the spectral trace and pixel wavelength assignments.

NIRCam subarrays used for TSGRISM observations always have their "bottom" edge located at the physical bottom edge of the detector and vary in size vertically. The source spectrum trace will always be centered somewhere near row 34 in the vertical direction (dispersion running parallel to rows) of the dispersed image. So the larger subarrays just result in a larger region of sky above the spectrum.

For TSGRISM, `extract_2d` always produces a cutout that is 64 pixels in height (cross-dispersion direction), regardless of whether the original image is full-frame or subarray. This cutout height is equal to the height of the smallest available subarray (2048 x 64). This is to allow area within the cutout for sampling the background in later steps, such as `extract_1d`. The slit height is a parameter that a user can set (during reprocessing) to tailor their results, but the entire extent of the image in the dispersion direction (along the image x-axis) is always included in the cutout.

The dispersion direction is horizontal for this mode, and it will be documented by copying the keyword "DISPAXIS" (with value 1) from the input file to the output cutout.

Step Arguments

The `extract_2d` step has various optional arguments that apply to certain observation modes. For NIRSpec observations there is one applicable argument:

`--slit_name`
: name [string value] of a specific slit region to extract. The default value of None will cause all known slits for the instrument mode to be extracted.

There are several arguments available for Wide-Field Slitless Spectroscopy (WFSS) and Time-Series (TSO) grism spectroscopy:

`--tsgrism_extract_height`
: int. The cross-dispersion extraction size, in units of pixels. Only applies to TSO mode.

`--wfss_extract_half_height`
: int. The cross-dispersion half size of the extraction region, in pixels, applied to point sources. Only applies to WFSS mode.

`--wfss_mmag_extract`
: float (default is `None`). The minimum (faintest) magnitude object to extract, based on the value of `isophotal_abmag` in the source catalog. Only applies to WFSS mode.

`--wfss_nbright`
: int (default is 1000). The number of brightest source catalog objects to extract. Can be used in conjunction with `wfss_mmag_extract`. Only applies to WFSS mode.

`--extract_orders`
: list. The list of spectral orders to extract. The default is taken from the `wavelengthrange` reference file. Applies to both WFSS and TSO modes.

`--grism_objects`
: list (default is empty). A list of `jwst.transforms.models.GrismObject`.

Reference Files

The `extract_2d` step uses the WAVELENGTHRANGE reference file. The WAVELENGTHRANGE reference file is only used for NIRCam and NIRISS Wide-Field Slitless Spectroscopy (WFSS) exposures.

WAVELENGTHRANGE Reference File

REFTYPE
WAVELENGTHRANGE

Data model
`WavelengthrangeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.WavelengthrangeModel.html#j

The WAVELENGTHRANGE reference file contains information on the minimum and maximum wavelengths of various spectroscopic modes, which can be order-dependent. The reference data are used to construct bounding boxes around the spectral traces produced by each object in the NIRCam and NIRISS WFSS modes. If a list of `GrismObject` is supplied, then no reference file is necessary.

Reference Selection Keywords for WAVELENGTHRANGE

CRDS selects appropriate WAVELENGTHRANGE references based on the following keywords. WAVELENGTHRANGE is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for WAVELENGTHRANGE

In addition to the standard reference file keywords listed above, the following keywords are *required* in WAVELENGTHRANGE reference files

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

These keywords are used as CRDS selectors

Reference Selection Keywords for WAVELENGTHRANGE

CRDS selects appropriate WAVELENGTHRANGE references based on the following keywords. WAVELENGTHRANGE is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Reference File Format

WAVELENGTHRANGE reference files are in ASDF format, with the format and contents specified by the `WavelengthrangeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.WavelengthrangeModel.html#jwst.datamoc data model schema. The exact content varies a bit depending on instrument mode, as explained in the following sections.

MIRI MRS

For MIRI MRS, the WAVELENGTHRANGE file consists of two fields that define the wavelength range for each combination of channel and band.

channels
: An ordered list of all possible channel and band combinations for MIRI MRS, e.g. "1SHORT".

wavelengthrange
: An ordered list of (lambda_min, lambda_max) for each item in the list above

NIRSpec

For NIRSpec, the WAVELENGTHRANGE file is a dictionary storing information about default wavelength range and spectral order for each combination of filter and grating.

filter_grating

order
Default spectral order

range
Default wavelength range

NIRCam WFSS, NIRCam TSGRISM, NIRISS WFSS

For NIRCam WFSS and TSGRIM modes, as well as NIRISS WFSS mode, the WAVELENGTHRANGE reference file contains the wavelength limits to use when calculating the minimum and maximum dispersion extents on the detector. It also contains the default list of orders that should be extracted for each filter. To be consistent with other modes, and for convenience, it also lists the orders and filters that are valid with the file.

order
A list of orders this file covers

wavelengthrange
A list containing the list of [order, filter, wavelength min, wavelength max]

waverange_selector
The list of FILTER names available

extract_orders
A list containing the list of orders to extract for each filter

jwst.extract_2d Package

Classes

Extract2dStep([name, parent, config_file, ...])	This Step performs a 2D extraction of spectra.

Extract2dStep

class jwst.extract_2d.**Extract2dStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

This Step performs a 2D extraction of spectra.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.

- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input_model, *args, **kwargs)	This is where real work happens.

Attributes Documentation

class_alias = 'extract_2d'

reference_file_types = ['wavelengthrange']

spec

```
slit_name = string(default=None)
extract_orders = int_list(default=None)  # list of orders to extract
grism_objects = list(default=None)  # list of grism objects to use
tsgrism_extract_height =  integer(default=None)  # extraction height in pixels,␣
↪TSGRISM mode
wfss_extract_half_height =  integer(default=5)  # extraction half height in␣
↪pixels, WFSS mode
wfss_mmag_extract = float(default=None)  # minimum abmag to extract, WFSS mode
wfss_nbright = integer(default=1000)  # number of brightest objects to extract,␣
↪WFSS mode
```

Methods Documentation

process(*input_model*, **args*, ***kwargs*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.20 FITS Generator

Description

Overview

The FITS generator is used to convert data from several different types of ground test data to DMS Level1b format data. This format is described in the document `DMS Level 1 and 2 Data Product Design - JWST-STScI-002111` by Daryl Swade. The code uses a collection of templates that govern the population of Level 1b header keyword values from the data in the input file headers, with different templates for different file types. The FITS generator will transform the input data (in detector coordinates) to the DMS coordinate system, where all of the imaging data has the same parity as the sky and very similar orientations.

Input details

To run the FITS generator, a 'proposal' file is required. There should be only one proposal file per directory, and it should have a name like

ddddd.prop

where d stands for a decimal digit. This file gives the names of each input FITS datafile, whether a subarray needs to be extracted from it and the exposure type (EXP_TYPE), as well as the relationship between the files from an operational viewpoint (i.e. Observation, Visit, ParallelSequenceID, Activity, Exposure, Detector). The file has a structure similar to XML with nested groups:

```
<Proposal title="MIRI FM IMG_OPT_01_FOV">
  <Observation>
    <Visit>
      <VisitGroup>
        <ParallelSequenceID>
          <Activity>
            <Exposure>
              <Detector>
```

(continues on next page)

(continued from previous page)

```
                <base>MIRFM1T00012942_1_493_SE_2011-07-13T10h45m00.fits</base>
                <subarray></subarray>
                <exp_type>MIR_IMAGE</exp_type>
              </Detector>
            </Exposure>
          </Activity>
        </ParallelSequenceID>
      </VisitGroup>
    </Visit>
  </Observation>
</Proposal>
```

Each nest can be repeated as needed. The <Detector></Detector> tags contain the information for each input/output file, with the input file name inside the <base> tags, the name of the subarray to be extracted within the <subarray> tag, and the exposure type within the <exp_type> tag.

The files within the <base> tag should be in the same directory as the proposal file.

The input FITS files can be from any of several different sources:

1. MIRI VM2 testing
2. MIRI FM testing
3. NIRSPEC FM testing
4. NIRSPEC IPS Simulator
5. NIRCAM NCONT testing (detector only)
6. NIRCAM FM testing
7. NIRISS CV testing
8. FGS CV testing

Most data that has been taken using the FITSWriter tool can be successfully converted to Level 1b format.

Command-line scripts

create_data directory

create_data followed by a directory will process the proposal file (generally a 5-digit string followed by '.prop') in that directory. The proposal file contains the names of the FITS files to be processed and the relationship between the exposures, allowing a unique numbering scheme.

Each FITS file referred to in the exposure will be processed to make a Level1b format JWST dataset with the pixel data flipped and/or rotated to make it conform to the DMS coordinate system, in which all imaging data has roughly the same orientation and parity on the sky.

The 5-digit string is used in the name of the Level 1b product, in that file 12345.prop will make data of the form

jw12345aaabbb_cccdd_eeeee_DATATYPE_uncal.fits.

The numbers that fill in the other letter spaces come from the structure of the proposal file, which is a sequence of nested levels. As each level is repeated, the number assigned to represent that level increments by 1.

Create_data Proposal File Format

The proposal file has an XML-like format that lays out the relationship between a set of exposures. The layout looks like this:

```
<Proposal title="Test">
  <Observation>
    <Visit>
      <VisitGroup>
        <ParallelSequenceID>
          <Activity>
            <Exposure>
              <Detector>
                <base></base>
                <subarray></subarray>
                <exp_type></exp_type>
              </Detector>
            </Exposure>
          </Activity>
        </ParallelSequenceID>
      </VisitGroup>
    </Visit>
  </Observation>
</Proposal>
```

The file to be converted is put between the <base></base> tags, and if a subarray is needed to be extracted from a full-frame exposure, the name of the subarray can be put between the <subarray></subarray> tags. Finally, the type of exposure can be placed between the <exp_type> </exp_type> tags. The values of EXP_TYPE are:

MIRI	NIRCAM	NIRSPEC	NIRISS	FGS
MIR_IMAGE	NRC_IMAGE	NRS_TASLIT	NIS_IMAGE	FGS_IMAGE
MIR_TACQ	NRC_TACQ	NRS_TACQ	NIS_FOCUS	FGS_FOCUS
MIR_LYOT	NRC_CORON	NRS_TACONFIRM	NIS_DARK	FGS_SKYFLAT
MIR_4QPM	NRC_FOCUS	NRS_CONFIRM	NIS_WFSS	FGS_INTFLAT
MIR_LRS-FIXEDSLIT	NRC_DARK	NRS_FIXEDSLIT		
MIR_LRS-SLITLESS	NRC_FLAT	NRS_AUTOWAVECAL		
MIR_MRS		NRS_IFU		
MIR_DARK		NRS_MSA		
MIR_FLAT		NRS_AUTOFLAT		
		NRS_DARK		
		NRS_LAMP		

Sections of this file can be replicated to represent, for example, all of the NIRCAM exposures from each of the 10 detectors at a single pointing by just replicating the <detector></detector> blocks.

Template file format

File types are described using a simple file format that vaguely resembles FITS headers.

Since it is necessary to create templates for several different flavors of data (FITSWriter, NIRSpec simulations, NIRCam homebrew etc) as well as different EXP_TYPEs that share many sections of data header but differ in other sections, the templates are divided into sections that are included. So a typical template for a particular flavor of data might look like this:

```
<<file nirspec_ifu_level1b>>
<<header primary>>
#include "level1b.gen.inc"
#include 'observation_identifiers.gen.inc'
#include 'exposure_parameters.gen.inc'
#include 'program_information.gen.inc'
#include 'observation_information.gen.inc'
#include 'visit_information.gen.inc'
#include 'exposure_information.gen.inc'
#include 'target_information.gen.inc'
#include 'exposure_times.gen.inc'
#include 'exposure_time_parameters.gen.inc'
#include 'subarray_parameters.gen.inc'
#include 'nirspec_configuration.gen.inc'
#include 'lamp_configuration.gen.inc'
#include 'guide_star_information.gen.inc'
#include 'jwst_ephemeris_information.gen.inc'
#include 'spacecraft_pointing_information.gen.inc'
#include 'aperture_pointing_information.gen.inc'
#include 'wcs_parameters.gen.inc'
#include 'velocity_aberration_correction.gen.inc'
#include 'nirspec_ifu_dither_pattern.gen.inc'
#include 'time_related.gen.inc'

<<data>>

<<header science>>
#include 'level1b_sci_extension_basic.gen.inc'

<<data>>
input[0].data.reshape((input[0].header['NINT'], \
                       input[0].header['NGROUP'], \
                       input[0].header['NAXIS2'], \
                       input[0].header['NAXIS1'])). \
                       astype('uint16')

<<header error>>
EXTNAME = 'ERR'

<<data>>
np.ones((input[0].header['NINT'], \
         input[0].header['NGROUP'], \
         input[0].header['NAXIS2'], \
         input[0].header['NAXIS1'])). \
         astype('float32')
```

(continues on next page)

(continued from previous page)

```
<<header data_quality>>
EXTNAME = "DQ"

<<data>>
np.zeros((input[0].header['NINT'], \
          input[0].header['NGROUP'], \
          input[0].header['NAXIS2'], \
          input[0].header['NAXIS1']), dtype='int16')
```

This has some regular generator syntax, but the bulk of the content comes from the `#include` directives.

By convention, templates have the extension `gen.txt`, while include files have the extension `inc`.

Basic syntax

Template files are in a line-based format.

Sections of the file are delimited with lines surrounded by << and >>. For example:

```
<<header primary>>
```

indicates the beginning of the primary header section.

Comments are lines beginning with #.

Lines can be continued by putting a backslash character (\) at the end of the line:

```
DETECTOR  = { 0x1e1: 'NIR', \
              0x1e2: 'NIR', \
              0x1ee: 'MIR', \
            }[input('SCA_ID')] / Detector type
```

Other files can be included using the include directive:

```
#include "other.file.txt"
```

Generator template

The generator template follows this basic structure:

- `file` line
- Zero or more HDUs, each of which has
  - a `header` section defining how keywords are generated
  - an optional `data` section defining how the data is converted

`file` line

The template must begin with a file line to give the file type a name. The name must be a valid Python identifier. For example:

```
<<file level1b>>
```

HDUs

Each HDU is defined in two sections, the header and data.

Header

The header begins with a header section line, giving the header a name, which must be a valid Python identifier. For example:

```
<<header primary>>
```

Following that is a list of keyword definitions. Each line is of the form:

```
KEYWORD = expression / comment
```

`KEYWORD` is a FITS keyword, may be up to 8 characters, and must contain only A through Z, _ and -.

The expression section is a Python expression that defines how the keyword value is generated. Within the namespace of the expression are the following:

- **Source functions:** Functions to retrieve keyword values from the input files. `input` gets values from the input FITS file, and there are any number of additional functions which get values from the input data files. For example, if the input data files include a file for program data, the function `program` is available to the expression that retrieves values from the program data file. If the function is provided with no arguments, it retrieves the value with the same key as the output keyword. If the function is provided with one argument, it is the name of the source keyword. For example:

  ```
  OBS_ID = input()
  ```

  copies the OBS_ID value from the corresponding HDU in the source FITS file to the OBS_ID keyword in the output FITS file. It is also possible to copy from a keyword value of a different name:

  ```
  CMPLTCND = input('CMPLTCON')
  ```

  To grab a value from the program data file, use the `program` function instead:

  ```
  TARGET = program()
  ```

- **Generator functions:** There are a number of helper functions in the `generators` module that help convert and generate values of different kinds. For example:

  ```
  END_TIME = date_and_time_to_cds(input('DATE-END'), input('TIME-END'))
  ```

  creates a CDS value from an input date and time.

- **Python expression syntax:** It's possible to do a lot of useful things just by using regular Python expression syntax. For example, to make the result a substring of a source keyword:

```
PARASEQN = input('OBS_ID')[13:14] / Parallel Sequence ID
```

or to calculate the difference of two values:

```
DURATION = input('END_TIME') - input('START_TIME')
```

The optional comment section following a / character will be attached to the keyword in the output FITS file. There is an important distinction between these comments which end up in the output FITS file, and comments beginning with # which are included in the template for informational purposes only and are ignored by the template parser.

It is also possible to include comments on their own lines to create section headings in the output FITS file. For example:

```
/ MIRI-specific keywords
FILTER    = '' / Filter element used
FLTSUITE  = '' / Flat field element used
WAVLNGTH  = '' / Wavelength requested in the exposure specification
GRATING   = '' / Grating/dichroic wheel position
LAMPON    = '' / Internal calibration lamp
CCCSTATE  = '' / Contamination control cover state

/ Exposure parameters
READPATT  = '' / Readout pattern
NFRAME    = 1 / Number of frames per read group
NSKIP     = 0 / Number of frames dropped
FRAME0    = 0 / zero-frame read
INTTIME   = 0 / Integration time
EXPTIME   = 0 / Exposure time
DURATION  = 0 / Total duration of exposure
OBJ_TYPE  = 'FAINT' / Object type
```

#include files will typically be just lines defining keyword definitions as above, for example, the file target_information.gen.inc looks like this:

```
/ Target information

TARGPROP = input('TARGNAME') / proposer's name for the target
TARGNAME = 'NGC 104' / standard astronomical catalog name for target
TARGTYPE = 'FIXED' / fixed target, moving target, or generic target
TARG_RA  = 0.0 / target RA computed at time of exposure
TARGURA  = 0.0 / target RA uncertainty
TARG_DEC = 0.0 / target DEC computed at time of exposure
TARRUDEC =  0.0  / target Dec uncertainty
PROP_RA  =  0.0  / proposer specified RA for the target
PROP_DEC =  0.0  / proposer specified Dec for the target
PROPEPOC = 2000.0  / proposer specified epoch for RA and Dec
```

and is used in many of the top-level level1b templates.

Data

The data section consists of a single expression that returns a Numpy array containing the output data.

The following are available in the namespace:

- `np`: `import numpy as np`
- `input`: A fits HDUList object containing the content of the input FITS file.
- `output`: A fits HDUList object containing the content of the output FITS file. Note that the output FITS file may only be partially constructed. Importantly, higher-number HDUs will not yet exist.

A complete example

```
# This file defines the structure of a MIRI level 1b file
<<file miri_level1b>>
<<header primary>>
SIMPLE    = T
BITPIX    = 32
NAXIS     = 0
EXTEND    = T
ORIGIN    = 'STScI'
TELESCOP  = 'JWST'
FILENAME  = '' / The filename
DATE      = now() / Date this file was generated

#include "level1a.gen.inc"

#include "level1b.gen.inc"

/ MIRI-specific keywords
FILTER    = '' / Filter element used
FLTSUITE  = '' / Flat field element used
WAVLNGTH  = '' / Wavelength requested in the exposure specification
GRATING   = '' / Grating/dichroic wheel position
LAMPON    = '' / Internal calibration lamp
CCCSTATE  = '' / Contamination control cover state

/ Exposure parameters
READPATT  = '' / Readout pattern
NFRAME    = 1 / Number of frames per read group
NSKIP     = 0 / Number of frames dropped
FRAME0    = 0 / zero-frame read
INTTIME   = 0 / Integration time
EXPTIME   = 0 / Exposure time
DURATION  = 0 / Total duration of exposure
OBJ_TYPE  = 'FAINT' / Object type

/ Subarray parameters
SUBARRAY  = '' / Name of subarray used
SUBXSTRT  = 0 / x-axis pixel number of subarray origin
SUBXSIZE  = 0 / length of subarray along x-axis
SUBTSTRT  = 0 / y-axis pixel number of subarray origin
```

(continues on next page)

(continued from previous page)

```
SUBYSIZE  = 0 / length of subarray along y-axis
LIGHTCOL  = 0 / Number of light-sensitive columns

<<data>>

<<header science>>
XTENSION  = 'IMAGE' /        FITS extension type
BITPIX    =         /        bits per data value
NAXIS     =         /        number of data array dimensions
NAXIS1    =         /        length of first data axis (#columns)
NAXIS2    =         /        length of second data axis (#rows)
NAXIS3    =         /        length of third data axis (#groups/integration)
NAXIS4    =         /        length of fourth data axis (#integrations)
PCOUNT    = 0       /        number of parameter bytes following data table
GCOUNT    = 1       /        number of groups
EXTNAME   = 'SCI'   /        extension name
BSCALE    = 1.0     /        scale factor for array value to physical value
BZERO     = 32768   /        physical value for an array value of zero
BUNIT     = 'DN'    /        physical units of the data array values

<<data>>
input[0].data.reshape((input[0].header['NINT'], \
                       input[0].header['NGROUP'], \
                       input[0].header['NAXIS2'], \
                       input[0].header['NAXIS1'])). \
                      astype('uint16')
```

jwst.fits_generator Package

15.1.21 First Frame Correction

Description

Class
jwst.firstframe.FirstFrameStep

Alias
firstframe

The MIRI first frame correction step flags the first group in every integration as bad (the "DO_NOT_USE" data quality flag is added to the GROUPDQ array), but only if the total number of groups per integration is greater than 3. This results in the data contained in the first group being excluded from subsequent steps, such as jump detection and ramp fitting. No flags are added if NGROUPS <= 3, because doing so would leave too few good groups to work with in later steps.

Only the GROUPDQ array is modified. The SCI, ERR, and PIXELDQ arrays are unchanged.

Step Arguments

The first frame correction has no step-specific arguments.

Reference File

This step does not use any reference file.

jwst.firstframe Package

Classes

FirstFrameStep([name, parent, config_file, ...])	FirstFrameStep: This is a MIRI specific task.

FirstFrameStep

class jwst.firstframe.**FirstFrameStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

FirstFrameStep: This is a MIRI specific task. If the number of groups is greater than 3, the DO_NOT_USE group data quality flag is added to first group.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'firstframe'

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.22 Flatfield Correction

Description

Class

jwst.flatfield.FlatFieldStep

Alias

flat_field

At its basic level this step flat-fields an input science dataset by dividing by a flat-field reference image. In particular, the SCI array from the flat-field reference file is divided into the SCI array of the science dataset, the flat-field DQ array is combined with the science DQ array using a bitwise OR operation, and variance and error arrays in the science dataset are updated to include the flat-field uncertainty. Details for particular modes are given in the sections below.

Upon completion of the step, the step status keyword "S_FLAT" gets set to "COMPLETE" in the output science data.

Imaging and Non-NIRSpec Spectroscopic Data

Simple imaging data, usually in the form of an ImageModel, and some spectroscopic modes, use a straight-forward approach that involves applying a single flat-field reference file to the science image. The spectroscopic modes included in this category are NIRCam WFSS and Time-Series Grism, NIRISS WFSS and SOSS, and MIRI MRS and LRS. All of these modes are processed as follows:

1. If the science data have been taken using a subarray and the FLAT reference file is a full-frame image, extract the corresponding subarray region from the flat-field data.
2. Find pixels that have a value of NaN or zero in the FLAT reference file SCI array and set their DQ values to "NO_FLAT_FIELD" and "DO_NOT_USE."
3. Reset the values of pixels in the flat that have DQ="NO_FLAT_FIELD" to 1.0, so that they have no effect when applied to the science data.
4. Propagate the FLAT reference file DQ values into the science exposure DQ array using a bitwise OR operation.
5. Apply the flat according to:

$$SCI_{science} = SCI_{science}/SCI_{flat}$$

$$VAR\_POISSON_{science} = VAR\_POISSON_{science}/SCI^2_{flat}$$

$$VAR\_RNOISE_{science} = VAR\_RNOISE_{science}/SCI^2_{flat}$$

$$VAR\_FLAT_{science} = (SCI^2_{science}/SCI^2_{flat}) * ERR^2_{flat}$$

$$ERR_{science} = \sqrt{VAR\_POISSON + VAR\_RNOISE + VAR\_FLAT}$$

Multi-integration datasets ("_rateints.fits" products), which are common for modes like NIRCam Time-Series Grism, NIRISS SOSS, and MIRI LRS Slitless, are handled by applying the above equations to each integration.

For guider exposures, the flat is applied in the same manner as given in the equations above, except for several differences. First, the variances due to Poisson noise and read noise are not calculated. Second, the output ERR array is the combined input ERR plus the flatfield ERR, summed in quadrature.

NIRSpec Spectroscopic Data

Flat-fielding of NIRSpec spectrographic data differs from other modes in that the flat-field array that will be applied to the science data is not read directly from CRDS. This is because the flat-field varies with wavelength and the wavelength of light that falls on any given pixel depends on the mode and which slits are open. The flat-field array that is divided into the SCI and ERR arrays is constructed on-the-fly by extracting the relevant section from the reference files, and then – for each pixel – interpolating to the appropriate wavelength for that pixel. This interpolation requires knowledge of the dispersion direction, which is read from keyword "DISPAXIS." See the Reference File section for further details.

For NIRSpec Fixed-Slit and MOS exposures, an on-the-fly flat-field is constructed to match each of the slits/slitlets contained in the science exposure. For NIRSpec IFU exposures, a single full-frame flat-field is constructed, which is applied to the entire science image.

NIRSpec NRS_BRIGHTOBJ data are processed just like NIRSpec Fixed-Slit data, except that NRS_BRIGHTOBJ data are stored in a CubeModel, rather than a MultiSlitModel. A 2-D flat-field image is constructed on-the-fly as usual, but this image is then divided into each plane of the 3-D science data arrays.

In all cases, there is a step option that allows for saving the on-the-fly flatfield to a file, if desired.

NIRSpec Fixed-Slit Primary Slit

The primary slit in a NIRSpec fixed-slit exposure receives special handling. If the primary slit, as given by the "FXD_SLIT" keyword value, contains a point source, as given by the "SRCTYPE" keyword, it is necessary to know the flatfield conversion factors for both a point source and a uniform source for use later in the *master background* step in Stage 3 processing. The point source version of the flatfield correction is applied to the slit data, but that correction is not appropriate for the background signal contained in the slit, and hence corrections must be applied later in the *master background* step.

So in this case the `flatfield` step will compute 2D arrays of conversion factors that are appropriate for a uniform source and for a point source, and store those correction factors in the "FLATFIELD_UN" and "FLATFIELD_PS" extensions, respectively, of the output data product. The point source correction array is also applied to the slit data.

Note that this special handling is only needed when the slit contains a point source, because in that case corrections to the wavelength grid are applied by the *wavecorr* step to account for any source mis-centering in the slit and the flatfield conversion factors are wavelength-dependent. A uniform source does not require wavelength corrections and hence the flatfield conversions will differ for point and uniform sources. Any secondary slits that may be included in a fixed-slit exposure do not have source centering information available, so the *wavecorr* step is not applied, and hence there's no difference between the point source and uniform source flatfield conversions for those slits.

Reference Files

The `flat_field` step uses four different types of reference files, depending on the type of data being processed. Most cases just use the FLAT reference file, while NIRSpec spectroscopic exposures use the three reference files FFLAT (fore optics), SFLAT (spectrograph optics), and DFLAT (detector).

FLAT Reference File

REFTYPE
FLAT

Data model
`FlatModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.FlatModel.html#jwst.datamodels.FlatModel)

The FLAT reference file contains pixel-by-pixel detector response values. It is used for all instrument modes except the NIRSpec spectroscopic modes.

Reference Selection Keywords for FLAT

CRDS selects appropriate FLAT references based on the following keywords. FLAT is not applicable for instruments not in the table. Non-standard keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, EXP_TYPE, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, FILTER, BAND, READPATT, SUBARRAY, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, FILTER, PUPIL, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, FILTER, PUPIL, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, FILTER, GRATING, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for FLAT

In addition to the standard reference file keywords listed above, the following keywords are *required* in FLAT reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for FLAT*):

Keyword	Data Model Name	Instruments
DETECTOR	model.meta.instrument.detector	All
EXP_TYPE	model.meta.exposure.type	FGS, NIRSpec
FILTER	model.meta.instrument.filter	MIRI, NIRCam, NIRISS, NIRSpec
PUPIL	model.meta.instrument.pupil	NIRCam, NIRISS
BAND	model.meta.instrument.band	MIRI
READPATT	model.meta.exposure.readpatt	MIRI
SUBARRAY	model.meta.subarray.name	MIRI
GRATING	model.meta.instrument.grating	NIRSpec

Reference File Format

FLAT reference files are FITS format, with 3 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float
ERR	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	integer
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The `DQ_DEF` table extension lists the bit assignments for the flag conditions used in the DQ array.

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

For application to imaging data, the FITS file contains a single set of SCI, ERR, DQ, and DQ_DEF extensions. Image dimensions should be 2048x2048 for the NIR detectors and 1032x1024 for MIRI (i.e. they include reference pixels), unless data were taken in subarray mode.

Reference Files for NIRSpec Spectroscopy

For NIRSpec spectroscopic data, the flat-field reference files allow for variations in the flat field with wavelength, as well as from pixel to pixel. There is a separate flat-field reference file for each of three sections of the instrument: the fore optics (FFLAT), the spectrograph (SFLAT), and the detector (DFLAT). The contents of the reference files differ from one mode to another (see below), but in general they may contain a flat-field image and a 1-D array. The image provides pixel-to-pixel values for the flat field that may vary slowly (or not at all) with wavelength, while the 1-D array is for a pixel-independent fast variation with wavelength. Details of the file formats are given in the following sections.

If there is no significant slow variation with wavelength, the image will be a 2-D array; otherwise, the image will be a 3-D array, with each plane corresponding to a different wavelength. In the latter case, the wavelength for each plane will be given in a table extension called WAVELENGTH in the flat-field reference file. The fast variation is given in a table extension called FAST_VARIATION, with column names "slit_name", "nelem", "wavelength", and "data" (an array of wavelength-dependent flat-field values). Each row of the table contains a slit name (for fixed-slit data, otherwise "ANY"), an array of flat-field values, an array of the corresponding wavelengths, and the number of elements ("nelem") of "data" and "wavelength" that are populated, because the allocated array size can be larger than needed. For some reference files there will not be any image array, in which case all the flat field information will be taken from the FAST_VARIATION table.

The SCI extension of the reference files may contain NaNs. If so, the `flat_field` step will replace these values with 1 and will flag the corresponding pixel in the DQ extension with NO_FLAT_FIELD. The WAVELENGTH extension is not expected to contain NaNs.

For the detector section, there is only one flat-field reference file for each detector. For the fore optics and the spectrograph sections, however, there are different flat fields for fixed-slit data, IFU data, and for multi-object spectroscopic data. Here is a summary of the contents of these files.

For the fore optics (FFLAT), the flat field for fixed-slit data contains just a FAST_VARIATION table (i.e. there is no image). This table has five rows, one for each of the fixed slits. The FFLAT for IFU data also contains just a FAST_VARIATION table, but it has only one row with the value "ANY" in the "slit_name" column. For multi-object spectroscopic data, the FFLAT contains four sets of images (one for each MSA quadrant), WAVELENGTH tables, and FAST_VARIATION tables. The images are unique to the FFLATs, however. The image "pixels" correspond to micro-shutter array slits, rather than to detector pixels. The array size is 365 columns by 171 rows, and there are multiple planes to handle the slow variation of flat field with wavelength.

For the spectrograph optics (SFLAT), the flat-field files have nearly the same format for fixed-slit data, IFU, and multi-object data. The difference is that for fixed-slit and IFU data, the image is just a single plane, i.e. the only variation with wavelength is in the FAST_VARIATION table, while there are multiple planes in the image for multi-object spectroscopic data (and therefore there is also a corresponding WAVELENGTH table, with one row for each plane of the image).

For the detector section, the DFLAT file contains a 3-D image (i.e. the flat field at multiple wavelengths), a corresponding WAVELENGTH table, and a FAST_VARIATION table with one row.

As just described, there are 3 types of reference files for NIRSpec (FFLAT, SFLAT, and DFLAT), and within each of these types, there are several formats, which are now described.

FFLAT Reference File

REFTYPE
FFLAT

There are 3 forms of NIRSpec FFLAT reference files: fixed slit, MSA spec, and IFU. For each type the primary HDU does not contain a data array.

Reference Selection Keywords for FFLAT

CRDS selects appropriate FFLAT references based on the following keywords. FFLAT is not applicable for instruments not in the table. Non-standard keywords used for file selection are *required*.

Instrument	Keywords
NIRSpec	INSTRUME, FILTER, EXP_TYPE, DATE-OBS, TIME-OBS

Fixed Slit

Data model
`NirspecFlatModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecFlatModel.html#jwst.datamo

The fixed slit FFLAT files have EXP_TYPE=NRS_FIXEDSLIT, and have a single BINTABLE extension, labeled FAST_VARIATION.

The table contains four columns:

- slit_name: string, name of slit
- nelem: integer, maximum number of wavelengths
- wavelength: float 1-D array, values of wavelength
- data: float 1-D array, flat field values for each wavelength

The number of rows in the table is given by NAXIS2, and each row corresponds to a separate slit.

MSA Spec

Data model
`NirspecQuadFlatModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecQuadFlatModel.html#j

The MSA Spec FFLAT files have EXP_TYPE=NRS_MSASPEC, and contain data pertaining to each of the 4 quadrants. For each quadrant, there is a set of 5 extensions - SCI, ERR, DQ, WAVELENGTH, and FAST_VARIATION. The file also contains a single DQ_DEF extension.

The extensions have the following characteristics:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	3	ncols x nrows x nelem	float
ERR	IMAGE	3	ncols x nrows x nelem	float
DQ	IMAGE	3	ncols x nrows x nelem	integer
WAVELENGTH	BINTABLE	2	TFIELDS = 1	N/A
FAST_VARIATION	BINTABLE	2	TFIELDS = 4	N/A
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

For the 5 extensions that appear multiple times, the EXTVER keyword indicates the quadrant number, 1 to 4. Each plane of the SCI array gives the throughput value for every shutter in the MSA quadrant for the corresponding wavelength, which is specified in the WAVELENGTH table. These wavelength-dependent values are combined with the FAST_VARIATION array, and are then applied to the science spectrum based on the wavelength of each pixel.

The WAVELENGTH table contains a single column:

- wavelength: float 1-D array, values of wavelength

Each of these wavelength values corresponds to a single plane of the IMAGE arrays.

The FAST_VARIATION table contains four columns:

- slit_name: the string "ANY"
- nelem: integer, maximum number of wavelengths
- wavelength: float 1-D array, values of wavelength
- data: float 1-D array, flat field values for each wavelength

The flat field values in this table are used to account for a wavelength-dependence on a much finer scale than given by the values in the SCI array. There is a single row in this table, which contains 1-D arrays of wavelength and flat-field values. The same wavelength-dependent value is applied to all pixels in a quadrant.

IFU

Data model
`NirspecFlatModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecFlatModel.html#jwst.datamo

The IFU FFLAT files have EXP_TYPE=NRS_IFU. These have one extension, a BINTABLE extension labeled FAST_VARIATION.

The FAST_VARIATION table contains four columns:

- slit_name: the string "ANY"
- nelem: integer, maximum number of wavelengths

- wavelength: float 1-D array, values of wavelength
- data: float 1-D array, flat field values for each wavelength

For each pixel in the science data, the wavelength of the light that fell on that pixel will be determined from the WAVELENGTH array in the science exposure (in the absence of that array, it will be computed using the WCS interface). The flat-field value for that pixel will then be obtained by interpolating within the wavelength and data arrays from the FAST_VARIATION table.

SFLAT Reference File

REFTYPE
SFLAT

Data model
`NirspecFlatModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecFlatModel.html#jwst.datamo

There are 3 types of NIRSpec SFLAT reference files: fixed slit, MSA spec, and IFU. For each type the primary HDU does not contain a data array.

Reference Selection Keywords for SFLAT

CRDS selects appropriate SFLAT references based on the following keywords. SFLAT is not applicable for instruments not in the table. Non-standard keywords used for file selection are *required*.

Instrument	Keywords
NIRSpec	INSTRUME, DETECTOR, FILTER, GRATING, EXP_TYPE, LAMP, OPMODE, DATE-OBS, TIME-OBS

Fixed Slit

The fixed slit references files have EXP_TYPE=NRS_FIXEDSLIT, and have a BINTABLE extension labeled FAST_VARIATION. The table contains four columns:

- slit_name: string, name of slit
- nelem: integer, maximum number of wavelengths
- wavelength: float 1-D array, values of wavelength
- data: float 1-D array, flat field values for each wavelength

The number of rows in the table is given by NAXIS2, and each row corresponds to a separate slit.

MSA Spec

The MSA Spec SFLAT files have EXP_TYPE=NRS_MSASPEC. They contain 6 extensions, with the following characteristics:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	3	ncols x nrows x n_wl	float
ERR	IMAGE	3	ncols x nrows x n_wl	float
DQ	IMAGE	3	ncols x nrows x n_wl	integer
WAVELENGTH	BINTABLE	2	TFIELDS = 1	N/A
FAST_VARIATION	BINTABLE	2	TFIELDS = 4	N/A
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

The keyword NAXIS3 in the 3 IMAGE extensions specifies the number, n_wl, of monochromatic slices, each of which gives the flat_field value for every pixel for the corresponding wavelength, which is specified in the WAVELENGTH table.

The WAVELENGTH table contains a single column:

- wavelength: float 1-D array, values of wavelength

Each of these wavelength values corresponds to a single plane of the IMAGE arrays.

The FAST_VARIATION table contains four columns:

- slit_name: the string “ANY”
- nelem: integer, maximum number of wavelengths
- wavelength: float 1-D array, values of wavelength
- data: float 1-D array, flat field values for each wavelength

The flat field values in this table are used to account for a wavelength-dependence on a much finer scale than given by the values in the SCI array. For each pixel in the science data, the wavelength of the light that fell on that pixel will be read from the WAVELENGTH array in the science exposure (if that array is absent, it will be computed using the WCS interface). The flat-field value for that pixel will then be obtained by interpolating within the wavelength and data arrays from the FAST_VARIATION table.

DFLAT Reference File

REFTYPE
DFLAT

Data model
`NirspecFlatModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecFlatModel.html#jwst.datamo

Reference Selection Keywords for DFLAT

CRDS selects appropriate DFLAT references based on the following keywords. DFLAT is not applicable for instruments not in the table. Non-standard keywords used for file selection are *required*.

Instrument	Keywords
NIRSpec	INSTRUME, DETECTOR, EXP_TYPE, DATE-OBS, TIME-OBS

There is one type of DFLAT reference file, containing 6 extensions with the following characteristics:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	3	ncols x nrows x n_wl	float
ERR	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	integer
WAVELENGTH	BINTABLE	2	TFIELDS = 1	N/A
FAST_VARIATION	BINTABLE	2	TFIELDS = 4	N/A
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

The keyword NAXIS3 in the SCI extension specifies the number, n_wl, of monochromatic slices, each of which gives the flat_field value for every pixel for the corresponding wavelength, which is specified in the WAVELENGTH table.

The WAVELENGTH table contains a single column:

- wavelength: float 1-D array, values of wavelength

Each of these wavelength values corresponds to a single plane of the SCI IMAGE array.

The FAST_VARIATION table contains four columns:

- slit_name: the string “ANY”
- nelem: integer, maximum number of wavelengths
- wavelength: float 1-D array, values of wavelength

- data: float 1-D array, flat field values for each wavelength

The flat field values in this table are used to account for a wavelength-dependence on a much finer scale than given by the values in the SCI array. There is a single row in this table, which contains 1-D arrays of wavelength and flat-field values. The same wavelength-dependent value is applied to all pixels in a quadrant.

Step Arguments

The `flat_field` step has the following optional arguments to control the behavior of the processing.

`--save_interpolated_flat` **(boolean, default=False)**
A flag to indicate whether to save to a file the NIRSpec flat field that was constructed on-the-fly by the step. Only relevant for NIRSpec data.

`--user_supplied_flat` **(string, default=None)**
The name of a user-supplied flat-field reference file.

`--inverse` **(boolean, default=False)**
A flag to indicate whether the math operations used to apply the flat-field should be inverted (i.e. multiply the flat-field into the science data, instead of the usual division).

jwst.flatfield Package

Classes

FlatFieldStep([name, parent, config_file, ...])	Flat-field a science image using a flatfield reference image.

FlatFieldStep

class jwst.flatfield.**FlatFieldStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Flat-field a science image using a flatfield reference image.

correction_pars

After the step has successfully run, the flat field applied is stored.

Type
{'flat': DataModel}

use_correction_pars

Use the flat stored in *correction_pars*

Type
boolean

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.

- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
flat_suffix	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.
skip_step(input_model)	Set the calibration switch to SKIPPED.

Attributes Documentation

class_alias = 'flat_field'

flat_suffix = 'interpolatedflat'

reference_file_types = ['flat', 'fflat', 'sflat', 'dflat']

spec

```
save_interpolated_flat = boolean(default=False) # Save interpolated NRS flat
user_supplied_flat = string(default=None)  # User-supplied flat
inverse = boolean(default=False)  # Invert the operation
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

skip_step(*input_model*)

Set the calibration switch to SKIPPED.

This method makes a copy of input_model, sets the calibration switch for the flat_field step to SKIPPED in the copy, closes input_model, and returns the copy.

Class Inheritance Diagram

15.1.23 Fringe Correction

Description

Class
jwst.fringe.FringeStep

Alias
fringe

The `fringe` step applies a fringe correction to MIRI MRS images. In particular, the SCI array from a fringe reference file is divided into the SCI and ERR arrays of the science data set. Only pixels that have valid (non-NaN) values in the SCI array of the reference file will be corrected. The DQ and variance arrays of the science exposure are not currently modified by this step.

The input to this step is in the form of an ImageModel data model. The fringe reference file that matches the input detector (MIRIFUSHORT or MIRIFULONG) and wavelength band (SHORT, MEDIUM, or LONG, as specified by GRATNG14) is used.

Upon successful application of this correction, the status keyword "S_FRINGE" is set to "COMPLETE".

Step Arguments

The `fringe` step has no step-specific arguments.

Reference Files

The `fringe` step uses a FRINGE reference file.

FRINGE Reference File

REFTYPE
FRINGE

Data model
`FringeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.FringeModel.html#jwst.datamodels.Fringe

The FRINGE reference file contains pixel-by-pixel fringing correction values.

Reference Selection Keywords for FRINGE

CRDS selects appropriate FRINGE references based on the following keywords. FRINGE is not applicable for instruments not in the table. Non-standard keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, DETECTOR, BAND, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for FRINGE

In addition to the standard reference file keywords listed above, the following keywords are *required* in FRINGE reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for FRINGE*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector
BAND	model.meta.instrument.band

Reference File Format

FRINGE reference files are FITS format, with 3 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float
ERR	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	integer
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The values in the SCI array give the correction values to be applied to the science data. Because MIRI MRS exposures are always full-frame, the image dimensions should be 1032 x 1024.

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

jwst.fringe Package

Classes

FringeStep([name, parent, config_file, ...])	FringeStep: Apply fringe correction to a science image using a fringe reference image.

FringeStep

class jwst.fringe.**FringeStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

FringeStep: Apply fringe correction to a science image using a fringe reference image.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.

- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'fringe'

reference_file_types = ['fringe']

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.24 Gain Scale Correction

Description

Class
jwst.gain_scale.GainScaleStep

Alias
gain_scale

The `gain_scale` step rescales pixel values in JWST countrate science data products in order to correct for the effect of using a non-standard detector gain setting. The countrate data are rescaled to make them appear as if they had been obtained using the standard gain setting.

This currently only applies to NIRSpec exposures that are read out using a subarray pattern, in which case a gain setting of 2 is used instead of the standard setting of 1. Note that this only applies to NIRSpec subarray data obtained after April 2017, which is when the change was made in the instrument flight software to use gain=2. NIRSpec subarray data obtained previous to that time used the standard gain=1 setting.

The `gain_scale` step is applied at the end of the *calwebb_detector1* pipeline, after the *ramp_fit* step has been applied. It is applied to both the "rate" and "rateints" products from *ramp_fit*, if both types of products were created. The science (SCI) and error (ERR) arrays are multiplied by the gain factor, and the Poisson variance (VAR_POISSON) and read noise variance (VAR_RNOISE) arrays are multiplied by the square of the gain factor.

The scaling factor is obtained from the "GAINFACT" keyword in the header of the gain reference file. Normally the *ramp_fit* step reads that keyword value during its execution and stores the value in the science data "GAINFACT" keyword, so that the gain reference file does not have to be loaded again by the `gain_scale` step. If, however, the step does not find that keyword populated in the science data, it loads the gain reference file to retrieve it. If all attempts to find the scaling factor fail, the step is skipped.

Gain reference files for instruments or modes that use the standard gain setting will typically not have the "GAINFACT" keyword in their header, which causes the `gain_scale` step to be skipped. Alternatively, gain reference files for modes that use the standard gain can have GAINFACT=1.0, in which case the correction is benign.

Upon successful completion of the step, the "S_GANSCL" keyword in the science data is set to "COMPLETE".

Arguments

The gain_scale correction has no step-specific arguments.

Reference File

The `gain_scale` step uses the GAIN reference file. It requires this reference file only to get the value of the "GAINFACT" keyword in the header of the file. This is the value used to rescale the science data. The `ramp_fit` step also uses the GAIN reference file and if it succeeded in finding the "GAINFACT" keyword when it was executed, it will have already stored the keyword value in the science data, for later use by the `gain_scale` step. In this case the `gain_scale` step will not read the GAIN reference file again when it runs.

GAIN reference file

REFTYPE
GAIN

Data model
`GainModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.GainModel.html#jwst.datamodels.GainModel

The GAIN reference file contains a pixel-by-pixel gain map, which can be used to convert pixel values from units of DN to electrons. The gain values are assumed to be in units of e/DN.

Reference Selection Keywords for GAIN

CRDS selects appropriate GAIN references based on the following keywords. GAIN is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for GAIN

In addition to the standard reference file keywords listed above, the following keywords are *required* in GAIN reference files, because they are used as CRDS selectors (see gain_selectors):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector
SUBARRAY	model.meta.subarray.name
BUNIT[1]	model.meta.bunit_data

[1] BUNIT is not used as a CRDS selector, but is required in the "SCI" extension header of GAIN reference files to document the units of the data. The expected value is "ELECTRONS/DN".

Reference File Format

GAIN reference files are FITS files with a single IMAGE extension. The FITS primary data array is assumed to be empty. The characteristics of the FITS extensions are as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float

jwst.gain_scale Package

Classes

GainScaleStep([name, parent, config_file, ...])	GainScaleStep: Rescales countrate data to account for use of a non-standard gain value.

GainScaleStep

class jwst.gain_scale.**GainScaleStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

GainScaleStep: Rescales countrate data to account for use of a non-standard gain value. All integrations are multiplied by the factor GAINFACT.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.

- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'gain_scale'

reference_file_types = ['gain']

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.25 Group Scale Correction

Description

Class
jwst.group_scale.GroupScaleStep

Alias
group_scale

The `group_scale` step rescales pixel values in raw JWST science data products to correct for instances where on-board frame averaging did not result in the proper downlinked values.

When multiple frames are averaged together on-board into a single group, the sum of the frames is computed and then the sum is divided by the number of frames to compute the average. Division by the number of frames is accomplished by simply bit-shifting the sum by an appropriate number of bits, corresponding to the decimal value of the number of frames. For example, when 2 frames are averaged into a group, the sum is shifted by 1 bit to achieve the equivalent of dividing by 2, and for 8 frames, the sum is shifted by 3 bits. The number of frames that are averaged into a group is recorded in the `NFRAMES` header keyword in science products and the divisor that was used is recorded in the `FRMDIVSR` keyword.

This method results in the correct average only when NFRAMES is a power of 2. When NFRAMES is not a power of 2, the next largest divisor is used to perform the averaging. For example, when NFRAMES=5, a divisor of 8 (bit shift of 3) is used to compute the average. This results in averaged values for each group that are too low by the factor NFRAMES/FRMDIVSR. This step rescales the pixel values by multiplying all groups in all integrations by the factor FRMDIVSR/NFRAMES.

The step decides whether rescaling is necessary by comparing the values of the NFRAMES and FRMDIVSR keywords. If they are equal, then the on-board averaging was computed correctly and this step is skipped. In this case, the calibration step status keyword `S_GRPSCL` is set to "SKIPPED." If the keyword values are not equal, rescaling is applied and the `S_GRPSCL` keyword is set to "COMPLETE".

It is assumed that this step is always applied to raw data before any other processing is done to the pixel values and hence rescaling is applied only to the SCI data array of the input product. It assumes that the ERR array has not yet been populated and hence there's no need for rescaling that array. The input GROUPDQ and PIXELDQ arrays are not affected by this step.

MIRI FASTGRPAVG mode

The MIRI detector readout pattern "FASTGRPAVG" results in individual frames being averaged together into a group, but the on-board averaging process is done differently than for other instruments. This results in a situation where the FRMDIVSR keyword gets assigned a value of 4, while NFRAMES still has a value of 1, despite the fact that 4 frames were actually averaged together to produce each downlinked group. This mismatch in keyword values would cause the `group_scale` step to think that rescaling needs to be applied.

To work around this issue, the original values of the number of frames per group and the number of groups per integration that are downlinked from the instrument are stored in the special keywords "MIRNFRMS" and "MIRNGRPS", respectively, so that their values are preserved. During Stage 1 processing in the pipeline, the value of the NFRAMES keyword is computed from MIRNFRMS * FRMDIVSR. The result is that when 4 frames are averaged together on board, both NFRAMES and FRMDIVSR will have a value of 4, which allows the `group_scale` step to correctly determine that no rescaling of the data is necessary.

Arguments

The `group_scale` correction has no step-specific arguments.

Reference File

The `group_scale` correction step does not use any reference files.

jwst.group_scale Package

Classes

GroupScaleStep([name, parent, config_file, ...])	GroupScaleStep: Rescales group data to account for on-board frame averaging that did not use FRMDIVSR = NFRAMES.

GroupScaleStep

class jwst.group_scale.**GroupScaleStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

GroupScaleStep: Rescales group data to account for on-board frame averaging that did not use FRMDIVSR = NFRAMES. All groups in the exposure are rescaled by FRMDIVSR/NFRAMES.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'group_scale'

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.26 Guider CDS Processing

Description

Class

jwst.guider_cds.GuiderCdsStep

Alias

guider_cds

The `guider_cds` step computes countrate images from the Correlated Double Sampling (CDS) detector readouts used in FGS guiding mode data. The exact way in which the countrate images are computed depends on the guiding mode (ID, ACQ1, ACQ2, TRACK, FineGuide) in use.

ID mode

The ID mode uses 2 integrations (NINTS=2) with 2 groups per integration (NGROUPS=2). For this mode the `guider_cds` step first computes a difference image for each integration by subtracting group 1 from group 2. A final difference image is then computed by taking the minimum value at each pixel from the 2 integrations. The minimum difference image is then divided by the group time to produce a countrate image. The output data array is 3D, with dimensions of (ncols x nrows x 1).

For this mode, the output ERR array has the same dimensions as the output data array. The values for the ERR array are calculated for each 2-group segment in each of the 2 integrations from the two variances of the slope of the segment.

The segment's variance due to read noise is:

$$var^R = \frac{2\,R^2}{tgroup^2}\,,$$

where R is the noise (using the default READNOISE pixel value) in the difference between the 2 groups and $tgroup$ is the group time in seconds (from the keyword TGROUP).

The segment's variance due to Poisson noise is:

$$var^P = \frac{slope}{tgroup \times gain}\,,$$

where $gain$ is the gain for the pixel (using the default GAIN pixel value), in e/DN. The $slope$ is the value of the pixel in the minimum difference image.

ACQ1, ACQ2, and TRACK modes

These modes use multiple integrations (NINTS>1) with 2 groups per integration (NGROUPS=2). For these modes the `guider_cds` step computes a countrate image for each integration, by subtracting group 1 from group 2 and dividing by the group time. The output data array is 3D, with dimensions of (ncols x nrows x nints).

For these modes, the values for the variances are calculated using the same equations as above for the ID mode, except :

1) $slope$ is the slope of the pixel.
2) R is the noise from the READNOISE reference file, or the default READNOISE pixel value if the reference file is not accessible.
3) $gain$ is the gain from the GAIN reference file, or the default GAIN pixel value if the reference file is not accessible.

FineGuide mode

The FineGuide mode uses many integrations (NINTS>>1) with 4 groups at the beginning and 4 groups at the end of each integration. The `guider_cds` step computes a countrate image for each integration by subtracting the average of the first 4 groups from the average of the last 4 groups and dividing by the group time. The output data array is 3D, with dimensions of (ncols x nrows x nints).

For this mode, the values for the variancesare calculated using the same equations as above for the ID mode, except $slope$ is the slope of the pixel, averaged over all integrations.

All modes

For all of the above modes, the square-root of the sum of the Poisson variance and read noise variance is written to the ERR extension.

After successful completion of the step, the "BUNIT" keyword in the output data is updated to "DN/s" and the "S_GUICDS" keyword is set to "COMPLETE".

Arguments

The `guider_cds` correction has no step-specific arguments.

Reference File

The `guider_cds` step uses two reference file types: GAIN and READNOISE.

Both the GAIN and READNOISE values are used to compute the total error estimates. If either reference file is inaccessible, representative default values will be used.

GAIN

READNOISE

jwst.guider_cds Package

Classes

GuiderCdsStep([name, parent, config_file, ...])	This step calculates the countrate for each pixel for FGS modes.

GuiderCdsStep

class jwst.guider_cds.**GuiderCdsStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

This step calculates the countrate for each pixel for FGS modes.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'guider_cds'

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.27 HLSP Processing

Description

Class
`jwst.coron.HlspStep`

Alias
hlsp

The `hlsp` step is one of the coronagraphic-specific steps in the `coron` sub-package. It produces high-level science products for KLIP-processed (PSF-subtracted) coronagraphic images. The step is currently a prototype and produces two simple products: a signal-to-noise ratio (SNR) image and a table of contrast data. The SNR image is computed by simply taking the ratio of the SCI and ERR arrays of the input target image. The contrast data are in the form of azimuthally-averaged noise versus radius. The noise is computed as the 1-sigma standard deviation within a set of concentric annuli centered in the input image. The annuli regions are computed to the nearest whole pixel; no sub-pixel calculations are performed.

Note: This step is not currently included in the *calwebb_coron3* pipeline, but can be run standalone.

Arguments

The `hlsp` step has one optional argument:

`--annuli_width`
: which is an integer parameter with a default value of 2 and is used to specify the width, in pixels, of the annuli to use when computing the contrast curve data.

Inputs

2D image

Data model
: `ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
: _psfsub

The input is the KLIP-processed (PSF-subtracted) image to be analyzed.

Outputs

2D SNR image

Data model
: `ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
: _snr

The computed SNR image.

Contrast table

Data model
: `ContrastModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ContrastModel.html#jwst.datamodels.C

File suffix
: _contrast

The table of contrast data, containing columns of radii (in pixels) and 1-sigma noise.

Reference Files

The `hlsp` step does not use any reference files.

jwst.coron.hlsp_step Module

Classes

HlspStep([name, parent, config_file, ...])	HlspStep: Make High-Level Science Products (HLSP's) from the results of coronagraphic exposure that's had KLIP processing applied to it.

HlspStep

class jwst.coron.hlsp_step.**HlspStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

HlspStep: Make High-Level Science Products (HLSP's) from the results of coronagraphic exposure that's had KLIP processing applied to it.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(target)	This is where real work happens.

Attributes Documentation

class_alias = 'hlsp'

spec

```
annuli_width = integer(default=2, min=1) # Width of contrast annuli
save_results = boolean(default=true) # Save results
```

Methods Documentation

process(*target*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.28 Imprint Subtraction

Description

Class

jwst.imprint.ImprintStep

Alias

imprint

The NIRSpec MSA imprint subtraction step removes patterns created in NIRSpec MOS and IFU exposures by the MSA structure. This is accomplished by subtracting a dedicated exposure taken with all MSA shutters closed and the IFU entrance aperture blocked.

The step has two input parameters: the target exposure and a list of one or more imprint exposures. These arguments can be provided as either file names or JWST data models.

In the event that multiple imprint images are provided, the step uses the meta data of the target and imprint exposures to find the imprint exposure that matches the observation number (keyword "OBSERVTN") and dither pattern position

number (keyword "PATT_NUM") of the target exposure. The matching imprint image is then subtracted from the target image. If no matching imprint image is found, the step will be skipped, returning the input target image unaltered.

When subtracting the imprint data model from the target data model, the SCI data array of the imprint exposure is subtracted from the SCI array of the target exposure, and the DQ arrays of the two exposures are combined using a bitwise logical OR operation. The ERR and variance arrays are not currently used or modified.

Step Arguments

The imprint subtraction step has no step-specific arguments.

Reference File

The imprint subtraction step does not use any reference files.

jwst.imprint Package

Classes

ImprintStep([name, parent, config_file, ...])	ImprintStep: Removes NIRSpec MSA imprint structure from an exposure by subtracting an imprint (a.k.a.

ImprintStep

class jwst.imprint.**ImprintStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

ImprintStep: Removes NIRSpec MSA imprint structure from an exposure by subtracting an imprint (a.k.a. leakcal) exposure.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input, imprint)	This is where real work happens.

Attributes Documentation

class_alias = 'imprint'

spec

Methods Documentation

process(*input*, *imprint*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.29 IPC Correction

Description

Class

jwst.ipc.IPCStep

Alias

ipc

The `ipc` step corrects a JWST exposure for interpixel capacitance by convolving with an IPC reference image.

The current implementation uses an IPC reference file that is normally a small, rectangular image (e.g. 3 x 3 pixels), a deconvolution kernel. The kernel may, however, be a 4-D array (e.g. 3 x 3 x 2048 x 2048), to allow the IPC correction to vary across the detector.

For each integration in the input science data, the data are corrected group-by-group by convolving with the kernel. Reference pixels are not included in the convolution; that is, their values will not be changed, and when the kernel overlaps a region of reference pixels, those pixels contribute a value of zero to the convolution. The ERR and DQ arrays will not be modified.

Subarrays

Subarrays are treated the same as full-frame data, with the exception that the reference pixels may be absent.

Step Arguments

The IPC deconvolution step has no step-specific arguments.

Reference Files

The IPC deconvolution step uses an IPC reference file.

IPC Reference File

REFTYPE
IPC

Data model
`IPCModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IPCModel.html#jwst.datamodels.IPCModel)

The IPC reference file contains a deconvolution kernel.

Reference Selection Keywords for IPC

CRDS selects appropriate IPC references based on the following keywords. IPC is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for IPC

In addition to the standard reference file keywords listed above, the following keywords are *required* in IPC reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for IPC*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector

Reference File Format

IPC reference files are FITS format, with 1 IMAGE extension. The FITS primary HDU does not contain a data array. The format and content of the file can be one of two forms, as described below:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	nkern x nkern	float
or				
SCI	IMAGE	4	ncols x nrows x nkern x nkern	float

Two formats are currently supported for the IPC kernel: a small 2-D array or a 4-D array. If the kernel is 2-D, its dimensions should be odd, for example 3 x 3 or 5 x 5 pixels. The value at the center pixel will be larger than 1 (e.g. 1.02533) and the sum of all pixel values will be equal to 1.

A 4-D kernel may be used to allow the IPC correction to vary from pixel to pixel across the image. In this case, the axes that are most rapidly varying (the last two in Python notation; the first two in IRAF/FITS notation) have dimensions equal to those of a full-frame image. At each point in that image, there will be a small, 2-D kernel as described in the previous paragraph.

jwst.ipc Package

Classes

IPCStep([name, parent, config_file, ...])	IPCStep: Performs IPC correction by convolving the input science data model with the IPC reference data.

IPCStep

class jwst.ipc.**IPCStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

IPCStep: Performs IPC correction by convolving the input science data model with the IPC reference data.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types

Methods Summary

process(input)	Apply the IPC correction.

Attributes Documentation

class_alias = 'ipc'

reference_file_types = ['ipc']

Methods Documentation

process(*input*)

Apply the IPC correction.

Parameters
input (*data model object*) – Science data model to be corrected.

Returns
IPC-corrected science data model.

Return type
data model object

Class Inheritance Diagram

15.1.30 Jump Detection

Description

Class
jwst.jump.JumpStep

Alias
jump

Assumptions

We assume that the `saturation` step has already been applied to the input science exposure, so that saturated values are appropriately flagged in the input GROUPDQ array. We also assume that steps such as the reference pixel correction (`refpix`) and non-linearity correction (`linearity`) have been applied, so that the input data ramps do not have any non-linearities or noise above the modeled Poisson and read noise due to instrumental effects. The absence of any of these preceding corrections or residual non-linearities or noise can lead to the false detection of jumps in the ramps, due to departure from linearity.

The `jump` step will automatically skip execution if the input data contain fewer than 3 groups per integration, because the baseline algorithm requires two first differences to work.

Algorithm

The algorithm for this step is called from the external package `stcal`, an STScI effort to unify common calibration processing algorithms for use by multiple observatories.

This routine detects jumps in an exposure by looking for outliers in the up-the-ramp signal for each pixel in each integration within an input exposure. On output, the GROUPDQ array is updated with the DQ flag "JUMP_DET" to indicate the location of each jump that was found. In addition, any pixels that have non-positive or NaN values in the gain reference file will have DQ flags "NO_GAIN_VALUE" and "DO_NOT_USE" set in the output PIXELDQ array. The SCI and ERR arrays of the input data are not modified.

The current implementation uses the two-point difference method described in Anderson & Gordon (2011) (https://ui.adsabs.harvard.edu/abs/2011PASP..123.1237A).

Two-Point Difference Method

The two-point difference method is applied to each integration as follows:

1. Compute the first differences for each pixel (the difference between adjacent groups)
2. Compute the clipped (dropping the largest difference) median of the first differences for each pixel.
3. Use the median to estimate the Poisson noise for each group and combine it with the read noise to arrive at an estimate of the total expected noise for each difference.
4. Compute the "difference ratio" as the difference between the first differences of each group and the median, divided by the expected noise.
5. If the largest "difference ratio" is greater than the rejection threshold, flag the group corresponding to that ratio as having a jump.
6. If a jump is found in a given pixel, iterate the above steps with the jump-impacted group excluded, looking for additional lower-level jumps that still exceed the rejection threshold.
7. Stop iterating on a given pixel when no new jumps are found or only one difference remains.
8. If the there are only three differences (four groups), the standard median is used rather than the clipped median.
9. If there are only two differences (three groups), the smallest one is compared to the larger one and if the larger one is above a threshold, it is flagged as a jump.
10. If flagging of the 4 neighbors is requested, then the 4 adjacent pixels will have ramp jumps flagged in the same group as the central pixel as long as it has a jump between the min and max requested levels for this option.
11. If flagging of groups after a ramp jump is requested, then the groups in the requested time since a detected ramp jump will be flagged as ramp jumps if the ramp jump is above the requested threshold. Two thresholds and times are possible for this option.

Note that any ramp values flagged as SATURATED in the input GROUPDQ array are not used in any of the above calculations and hence will never be marked as containing a jump.

Multiprocessing

This step has the option of running in multiprocessing mode. In that mode it will split the input data cube into a number of row slices based on the number of available cores on the host computer and the value of the max_cores input parameter. By default the step runs on a single processor. At the other extreme if max_cores is set to 'all', it will use all available cores (real and virtual). Testing has shown a reduction in the elapsed time for the step proportional to the number of real cores used. Using the virtual cores also reduces the elapsed time but at a slightly lower rate than the real cores.

If multiprocessing is requested the input cube will be divided into a number of slices in the row dimension (with the last slice being slightly larger, if needed). The slices are then sent to twopoint_difference.py by detect_jumps. After all the slices have finished processing, detect_jumps assembles the output group_dq cube from the slices.

Subarrays

The use of the reference files is flexible. Full-frame reference files can be used for all science exposures, in which case subarrays will be extracted from the reference file data to match the science exposure, or subarray-specific reference files may be used.

Large Events (Snowballs and Showers)

All the detectors on JWST are affected by large cosmic ray events. While these events, in general, affect a large number of pixels, the more distinguishing characteristic is that they are surrounded by a halo of pixels that have a low level of excess counts. These excess counts are, in general, below the detection threshold of normal cosmic rays.

To constrain the effect of this halo, the jump step will fit ellipses or circles that enclose the large events and expand the ellipses and circles by the input expansion_factor and mark them as jump.

The two types of detectors respond differently. The large events in the near-infrared detectors are almost always circles with a central region that is saturated. The saturated core allows the search for smaller events without false positives. The MIRI detectors do not, in general, have a saturated center and are only rarely circular. Thus, we fit the minimum enclosing ellipse and do not require that there are saturated pixels within the ellipse.

Arguments

The `jump` step has 30 optional arguments that can be set by the user:

Parameters for Baseline Cosmic Ray Jump Detection

- `--rejection_threshold`: A floating-point value that sets the sigma threshold for jump detection. In the code, sigma is determined using the read noise from the read noise reference file and the Poisson noise (based on the median difference between samples and the gain reference file). Note that any noise source beyond these two that may be present in the data will lead to an increase in the false positive rate and thus may require an increase in the value of this parameter. The default value of 4.0 for the rejection threshold will yield 6200 false positives for every million pixels, if the noise model is correct.
- `--three_group_rejection_threshold`: Cosmic ray sigma rejection threshold for ramps having 3 groups. This is a floating-point value with default value of 6.0, and minimum of 0.0.
- `--four_group_rejection_threshold`: Cosmic ray sigma rejection threshold for ramps having 4 groups. This is a floating-point value with default value of 5.0, and minimum of 0.0.
- `--maximum_cores`: The number of available cores that will be used for multi-processing in this step. The default value is '1', which does not use multi-processing. The other options are either an integer, 'quarter', 'half', or 'all'. Note that these fractions refer to the total available cores and on most CPUs these include physical and virtual cores. The clock time for the step is reduced almost linearly by the number of physical cores used on all machines.

For example, on an Intel CPU with six real cores and six virtual cores, setting maximum_cores to 'half' results in a decrease of a factor of six in the clock time for the step to run. Depending on the system, the clock time can also decrease even more with maximum_cores is set to 'all'. Setting the number of cores to an integer can be useful when running on machines with a large number of cores where the user is limited in how many cores they can use. Note that, currently, snowball and shower detection does not use multiprocessing.

- `--flag_4_neighbors`: If set to True (default is True) it will cause the four perpendicular neighbors of all detected jumps to also be flagged as a jump. This is needed because of the inter-pixel capacitance (IPC), which causes a small jump in the neighbors. The small jump might be below the rejection threshold, but will affect the slope determination of the pixel. The step will take about 40% longer to run when this is set to True.
- `--max_jump_to_flag_neighbors`: A floating point value in units of sigma that limits the flagging of neighbors. Any jump above this cutoff will not have its neighbors flagged. The concept is that the jumps in neighbors will be above the rejection threshold and thus be flagged as primary jumps. The default value is 200.
- `--min_jump_to_flag_neighbors`: A floating point value in units of sigma that limits the flagging of neighbors of marginal detections. Any primary jump below this value will not have its neighbors flagged. The goal is to prevent flagging jumps that would be too small to significantly affect the slope determination. The default value is 10.

Parameters that affect after jump Flagging

After a jump of at least 'after_jump_flag_dn1' DN, groups up to 'after_jump_flag_time1' seconds will also be flagged as jumps. That pair of arguments is defined as:

- `--after_jump_flag_dn1`: A floating point value in units of DN
- `--after_jump_flag_time1`: A floating point value in units of seconds

A second threshold and time can also be set: after a jump of at least 'after_jump_flag_dn2' DN, groups up to 'after_jump_flag_time2' seconds will also be flagged as jumps. That pair of arguments is defined as:

- `--after_jump_flag_dn2`: A floating point value in units of DN
- `--after_jump_flag_time2`: A floating point value in units of seconds

Parameters that affect Near-IR Snowball Flagging

- `--expand_large_events`: A boolean parameter that controls whether the jump step will expand the number of pixels that are flagged around large cosmic ray events. These are know as "snowballs" in the near-infrared detectors and "showers" for the MIRI detectors. In general, this should be set to True.
- `--min_jump_area`: The minimum number of contiguous pixels needed to trigger the expanded flagging of large cosmic rays events.
- `--min_sat_area`: The minimum number of saturated pixels required to meet "sat_required_snowball".
- `--expand_factor`: A multiplicative factor applied to the enclosing ellipse for snowballs. This larger area will have all pixels flagged as having a jump.
- `--use_ellipses`: deprecated
- `--sat_required_snowball`: A boolean value that if True requires that there are saturated pixels within the enclosed jump circle.
- `--min_sat_radius_extend`: The minimum radius of the saturated core of a snowball required to for the radius of the saturated core to be extended.
- `--sat_expand`: Number of pixels to add to the radius of the saturated core of snowballs
- `--edge_size`: The distance from the edge of the detector where saturated cores are not required for snowball detection

Parameters that affect MIRI Shower Flagging

- `--find_showers`: Turn on the detection of showers for the MIRI detectors
- `--extend_snr_threshold`: The SNR minimum for the detection of faint extended showers in MIRI
- `--extend_min_area`: The required minimum area of extended emission after convolution for the detection of showers in MIRI
- `--extend_inner_radius`: The inner radius of the ring_2D_kernel that is used for the detection of extended emission in showers
- `--extend_outer_radius`: The outer radius of the Ring2DKernal that is used for the detection of extended emission in showers
- `--extend_ellipse_expand_ratio`: Multiplicative factor to expand the radius of the ellipse fit to the detected extended emission in MIRI showers
- `--time_masked_after_showers`: Number of seconds to flag groups as jump after a detected extended emission in MIRI showers

Parameter that affects both Snowball and Shower flagging

- `--max_extended_radius`: The maxiumum extension of the jump and saturation that will be flagged for showers or snowballs

Parameters that affect Sigma Clipping

- `--minimum_groups`: The minimum number of groups to run the jump step with sigma clipping
- `--minimum_sigclip_groups`: The minimum number of groups to switch the jump detection to use sigma clipping
- `--only_use_ints`: If true the sigma clipping is applied only for a given group across all ints. If not, all groups from all ints are used for the sigma clipping.

Reference File Types

The `jump` step uses two reference files: GAIN and READNOISE. The GAIN reference file is used to temporarily convert pixel values in the `jump` step from units of DN to electrons. The READNOISE reference file is used in estimating the expected noise in each pixel. Both are necessary for proper computation of noise estimates within the `jump` step.

GAIN

READNOISE

jwst.jump Package

Classes

JumpStep([name, parent, config_file, ...])	JumpStep: Performs CR/jump detection on each ramp integration within an exposure.

JumpStep

class jwst.jump.**JumpStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

JumpStep: Performs CR/jump detection on each ramp integration within an exposure. The 2-point difference method is applied.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'jump'

reference_file_types = ['gain', 'readnoise']

spec

```
rejection_threshold = float(default=4.0,min=0) # CR sigma rejection threshold
three_group_rejection_threshold = float(default=6.0,min=0) # CR sigma rejection␣
↪threshold
four_group_rejection_threshold = float(default=5.0,min=0) # CR sigma rejection␣
↪threshold
```

(continues on next page)

(continued from previous page)

```
maximum_cores = string(default='1') # cores for multiprocessing. Can be an␣
↪integer, 'half', 'quarter', or 'all'
flag_4_neighbors = boolean(default=True) # flag the four perpendicular␣
↪neighbors of each CR
max_jump_to_flag_neighbors = float(default=1000) # maximum jump sigma that will␣
↪trigger neighbor flagging
min_jump_to_flag_neighbors = float(default=10) # minimum jump sigma that will␣
↪trigger neighbor flagging
after_jump_flag_dn1 = float(default=0) # 1st flag groups after jump above DN␣
↪threshold
after_jump_flag_time1 = float(default=0) # 1st flag groups after jump groups␣
↪within specified time
after_jump_flag_dn2 = float(default=0) # 2nd flag groups after jump above DN␣
↪threshold
after_jump_flag_time2 = float(default=0) # 2nd flag groups after jump groups␣
↪within specified time
expand_large_events = boolean(default=False) # Turns on Snowball detector for␣
↪NIR detectors
min_sat_area = float(default=1.0) # minimum required area for the central␣
↪saturation of snowballs
min_jump_area = float(default=5.0) # minimum area to trigger large events␣
↪processing
expand_factor = float(default=2.0) # The expansion factor for the enclosing␣
↪circles or ellipses
use_ellipses = boolean(default=False) # deprecated
sat_required_snowball = boolean(default=True) # Require the center of snowballs␣
↪to be saturated
min_sat_radius_extend = float(default=2.5) # The min radius of the sat core to␣
↪trigger the extension of the core
sat_expand = integer(default=2) # Number of pixels to add to the radius of the␣
↪saturated core of snowballs
edge_size = integer(default=25) # Size of region on the edges of NIR detectors␣
↪where a sat core is not required
find_showers = boolean(default=False) # Turn on shower flagging for MIRI
extend_snr_threshold = float(default=1.2) # The SNR minimum for detection of␣
↪extended showers in MIRI
extend_min_area = integer(default=90) # Min area of emission after convolution␣
↪for the detection of showers
extend_inner_radius = float(default=1) # Inner radius of the ring_2D_kernel␣
↪used for convolution
extend_outer_radius = float(default=2.6) # Outer radius of the ring_2D_Kernel␣
↪used for convolution
extend_ellipse_expand_ratio = float(default=1.1) # Expand the radius of the␣
↪ellipse fit to the extended emission
time_masked_after_shower = float(default=15) # Seconds to flag as jump after a␣
↪detected extended emission
max_extended_radius = integer(default=200) # The maximum radius of an extended␣
↪snowball or shower
minimum_groups = integer(default=3) # The minimum number of groups to perform␣
↪jump detection using sigma clipping
minimum_sigclip_groups = integer(default=100) # The minimum number of groups to␣
↪switch to sigma clipping
```

(continues on next page)

(continued from previous page)

```
only_use_ints = boolean(default=True) # In sigclip only compare the same group␣
↪across ints, if False compare all groups
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.31 KLIP Processing

Description

Class
`jwst.coron.KlipStep`

Alias
klip

The `klip` step is one of the coronagraphic-specific steps in the `coron` sub-package and is used in Stage 3 *calwebb_coron3* processing. It applies the Karhunen-Loeve Image Plane (KLIP) algorithm to coronagraphic images, using an accompanying set of reference PSF images, in order to fit and subtract an optimal PSF from a source image. The KLIP algorithm uses a KL decomposition of the set of reference PSF's, and generates a model PSF from the projection of the target on the KL vectors. The model PSF is then subtracted from the target image (Soummer, Pueyo, and Larkin 2012). KLIP is a Principle Component Analysis (PCA) method and is very similar to the Locally Optimized Combination of Images (LOCI) method. The main advantages of KLIP over LOCI are the possibility of direct forward modeling and a significant speed increase.

The KLIP algorithm consists of the following high-level steps:

1) Partition the target and reference PSF images in a set of search areas, and subtract their average values so that they have zero mean
2) Compute the KL transform of the set of reference PSF's
3) Choose the number of modes to keep in the estimated target PSF
4) Compute the best estimate of the target PSF from the projection of the target image on the KL eigenvectors
5) Calculate the PSF-subtracted target image

Arguments

The `klip` step has one optional argument:

`--truncate`
: This is an integer parameter with a default value of 50 and is used to specify the number of KL transform rows to keep when computing the PSF fit to the target.

Inputs

The `klip` step takes two inputs: a science target exposure in the form of a 3D data cube and a 4D aligned PSF image ("_psfalign") product.

3D calibrated images

Data model
: `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
: _calints

A 3D calibrated science target product containing a stack of per-integration images. This should be a "_calints" product created by the *calwebb_image2* pipeline. Normally one of the science target exposures specified in the ASN file used as input to the *calwebb_coron3* pipeline.

4D aligned PSF images

Data model
: `QuadModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.QuadModel.html#jwst.datamodels.QuadMod

File suffix
: _psfalign

A 4D collection of PSF images that have been aligned to each of the per-integration images contained in the science target "_calints" product, created by the *align_refs* step.

Outputs

3D PSF-subtracted images

Data model
: `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
: _psfsub

The output is a 3D stack of PSF-subtracted images of the science target, having the same dimensions as the input science target ("_calints") product. The PSF fitting and subtraction has been applied to each integration image independently. The file name syntax is exposure-based, using the root of the input "_calints" product, with the addition of the association candidate ID and the "_psfsub" product type suffix, e.g. "jw8607342001_02102_00001_nrcb3_a3001_psfsub.fits."

Reference Files

The `klip` step does not use any reference files.

jwst.coron.klip_step Module

Classes

KlipStep([name, parent, config_file, ...])	KlipStep: Performs KLIP processing on a science target coronagraphic exposure.

KlipStep

class jwst.coron.klip_step.**KlipStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

KlipStep: Performs KLIP processing on a science target coronagraphic exposure. The input science exposure is assumed to be a fully calibrated level-2b image. The processing is performed using a set of reference PSF images observed in the same coronagraphic mode.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(target, psfrefs)	This is where real work happens.

Attributes Documentation

class_alias = 'klip'

spec

```
truncate = integer(default=50,min=0) # The number of KL transform rows to keep
```

Methods Documentation

process(*target*, *psfrefs*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.32 Library Utilities

Engineering Database Interface

jwst.lib.engdb_tools Module

Access the JWST Engineering Mnemonic Database

The engineering mnemonics are provided by multiple services, all of which require a level of authentication.

For non-operational use, the providing service is through the MAST AUI website

https://mast.stsci.edu/portal/Mashup/Clients/jwstedb/jwstedb.html

Authorization can be requested through

https://auth.mast.stsci.edu/

Interface

The primary entry point is the function *jwst.lib.engdb_tools.ENGDB_Service*. This function returns a *jwst.lib.engdb_lib.EngdbABC* connection object. Using this object, values for a mnemonic covering a specified time range can be retrieved using the *get_values* method.

By default, only values inclusively between the time end points are returned. Depending on the frequency a mnemonic is updated, there can be no values. If values are always desired, the nearest, bracketing values outside the time range can be requested.

Warning: Many mnemonics are updated very quickly, up to 16Hz. When in doubt, specify a very short time frame, and request bracketing values. Otherwise, the request can return a very large amount of data, risking timeout, unnecessary memory consumption, or access restrictions.

Examples

The typical workflow is as follows:

```
from jwst.lib.engdb_tools import ENGDB_Service

service = ENGDB_Service()  # By default, will use the public MAST service.

values = service.get_values('sa_zattest2', '2021-05-22T00:00:00', '2021-05-22T00:00:01')
```

Environmental Variables

ENG_BASE_URL
: If no URL is specified in code or by command line parameters, this value is used. If not defined, a default, as defined by the individual services, will be attempted.

MAST_API_TOKEN
: If no token is provided in code or by command line parameters, this value will be used. *EngdbMast* service requires a token to be provided. See https://auth.mast.stsci.edu/ for more information.

ENG_RETRIES
: Number of attempts to make when connecting to the service. Default is 10.

ENG_TIMEOUT
: Number of seconds before timing out a network connection. Default is 600 seconds (10 minutes)

Functions

ENGDB_Service([base_url])	Access the JWST Engineering Database

ENGDB_Service

jwst.lib.engdb_tools.**ENGDB_Service**(*base_url=None*, ***service_kwargs*)

Access the JWST Engineering Database

Access can be either through the public MAST API or by direct connection to the database server.

Parameters

- **base_url** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None.*) – The base url for the engineering RESTful service
- **service_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Service-specific keyword arguments. Refer to the concrete implementations of *EngdbABC*.

Returns

service – The engineering database service to use.

Return type

EngdbABC

jwst.lib.engdb_mast Module

Access the JWST Engineering Mnemonic Database through MAST

Classes

EngdbMast([base_url, token])	Access the JWST Engineering Database through MAST

EngdbMast

class jwst.lib.engdb_mast.**EngdbMast**(*base_url=None*, *token=None*, ***service_kwargs*)

Bases: *EngdbABC*

Access the JWST Engineering Database through MAST

Parameters

- **base_url** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The base url for the engineering RESTful service. If not defined, the environmental variable ENG_BASE_URL is queried. Otherwise the default MAST website is used.
- **token** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – The MAST access token. If not defined, the environmental variable MAST_API_TOKEN is queried. A token is required. For more information, see 'https://auth.mast.stsci.edu/'
- **service_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Service-specific keyword arguments that are not relevant to this implementation of EngdbABC.

Raises

RuntimeError (https://docs.python.org/3/library/exceptions.html#RuntimeError) – Any and all failures with connecting with the MAST server.

Attributes Summary

base_url	The base URL for the engineering service.
endtime	The end time of the last query.
response	The results of the last query.
retries	Number of retries to attempt to contact the service
starttime	The start time of the last query.
timeout	Network timeout when communicating with the service
token	MAST Token

Methods Summary

cache(mnemonics, starttime, endtime, cache_path)	Cache results for the list of mnemonics
cache_as_local(mnemonics, starttime, ...)	Cache results for the list of mnemonics, but in the EngdbLocal format
configure([base_url, token])	Configure from parameters and environment
get_meta(*kwargs)	Get the mnemonics meta info
get_values(mnemonic, starttime, endtime[, ...])	Retrieve all results for a mnemonic in the requested time range.
set_session()	Setup HTTP session

Attributes Documentation

base_url = None

The base URL for the engineering service.

endtime = None

The end time of the last query.

response = None

The results of the last query.

retries = 10

Number of retries to attempt to contact the service

starttime = None

The start time of the last query.

timeout = 600

Network timeout when communicating with the service

token = None

MAST Token

Methods Documentation

cache(*mnemonics*, *starttime*, *endtime*, *cache_path*)

Cache results for the list of mnemonics

Parameters

- **mnemonics** (*iterable*) – List of mnemonics to retrieve
- **starttime** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or astropy.time.Time*) – The, inclusive, start time to retrieve from.
- **endtime** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or astropy.time.Time*) – The, inclusive, end time to retrieve from.
- **cache_path** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or Path-like*) – Path of the cache directory.

cache_as_local(*mnemonics*, *starttime*, *endtime*, *cache_path*)

Cache results for the list of mnemonics, but in the EngdbLocal format

The target format is native to what the EngdbDirect service provides.

Parameters

- **mnemonics** (*iterable*) – List of mnemonics to retrieve
- **starttime** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or astropy.time.Time*) – The, inclusive, start time to retrieve from.
- **endtime** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or astropy.time.Time*) – The, inclusive, end time to retrieve from.
- **cache_path** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or Path-like*) – Path of the cache directory.

configure(*base_url=None*, *token=None*)

Configure from parameters and environment

Parameters

- **base_url** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The base url for the engineering RESTful service. If not defined, the environmental variable ENG_BASE_URL is queried. Otherwise the default MAST website is used.
- **token** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – The MAST access token. If not defined, the environmental variable MAST_API_TOKEN is queried. A token is required. For more information, see 'https://auth.mast.stsci.edu/'

get_meta(**kwargs*)

Get the mnemonics meta info

The MAST interface does not provide any meta.

get_values(*mnemonic*, *starttime*, *endtime*, *time_format=None*, *include_obstime=False*, *include_bracket_values=False*, *zip_results=True*)

Retrieve all results for a mnemonic in the requested time range.

Parameters

- **mnemonic** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The engineering mnemonic to retrieve
- **starttime** (str or `astropy.time.Time`) – The, inclusive, start time to retrieve from.

- **endtime** (str or `astropy.time.Time`) – The, inclusive, end time to retrieve from.
- **time_format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format of the input time used if the input times are strings. If None, a guess is made.
- **include_obstime** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `True` (https://docs.python.org/3/library/constants.html#True), the return values will include observation time as `astropy.time.Time`. See `zip_results` for further details.
- **include_bracket_values** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – The DB service, by default, returns the bracketing values outside of the requested time. If `True` (https://docs.python.org/3/library/constants.html#True), include these values.
- **zip_results** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `True` (https://docs.python.org/3/library/constants.html#True) and `include_obstime` is `True` (https://docs.python.org/3/library/constants.html#True), the return values will be a list of 2-tuples. If false, the return will be a single 2-tuple, where each element is a list.

Returns

values – Returns the list of values. See `include_obstime` and `zip` (https://docs.python.org/3/library/functions.html#zip) for modifications.

Return type

[value, ...] or [(obstime, value), ...] or ([obstime,...], [value, ...])

set_session()

Setup HTTP session

jwst.lib.engdb_direct Module

Access the JWST Engineering Mnemonic Database through direct connection

Classes

EngdbDirect([base_url, default_format])	Access the JWST Engineering Database through direct connection

EngdbDirect

class jwst.lib.engdb_direct.**EngdbDirect**(*base_url=None*, *default_format='dict'*, ***service_kwargs*)

Bases: *EngdbABC*

Access the JWST Engineering Database through direct connection

Parameters

- **base_url** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The base url for the engineering RESTful service
- **default_format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format the results of the data should be returned from the service. If 'dict', the result will be in Python dict format.
- **service_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Service-specific keyword arguments that are not relevant to this implementation of EngdbABC.

Attributes Summary

base_url	The base URL for the engineering service.
default_format	The format the results of the data should be returned from the service.
endtime	The end time of the last query.
response	The results of the last query.
starttime	The start time of the last query.

Methods Summary

configure([base_url])	Configure from parameters and environment
get_meta([mnemonic, result_format])	Get the mnemonics meta info
get_values(mnemonic, starttime, endtime[, ...])	Retrieve all results for a mnemonic in the requested time range.
set_session()	Setup HTTP session

Attributes Documentation

base_url = None

The base URL for the engineering service.

default_format

The format the results of the data should be returned from the service.

endtime = None

The end time of the last query.

response = None

The results of the last query.

starttime = None

The start time of the last query.

Methods Documentation

configure(*base_url=None*)

Configure from parameters and environment

Parameters

base_url (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The base url for the engineering RESTful service

get_meta(*mnemonic=''*, *result_format=None*)

Get the mnemonics meta info

Parameters

- **mnemonic** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The engineering mnemonic to retrieve

- **result_format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format to request from the service. If None, the *default_format* is used.

get_values(*mnemonic*, *starttime*, *endtime*, *time_format=None*, *include_obstime=False*, *include_bracket_values=False*, *zip_results=True*)

Retrieve all results for a mnemonic in the requested time range.

Parameters

- **mnemonic** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The engineering mnemonic to retrieve
- **starttime** (str or `astropy.time.Time`) – The, inclusive, start time to retrieve from.
- **endtime** (str or `astropy.time.Time`) – The, inclusive, end time to retrieve from.
- **time_format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format of the input time used if the input times are strings. If None, a guess is made.
- **include_obstime** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `True` (https://docs.python.org/3/library/constants.html#True), the return values will include observation time as `astropy.time.Time`. See `zip_results` for further details.
- **include_bracket_values** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – The DB service, by default, returns the bracketing values outside of the requested time. If `True` (https://docs.python.org/3/library/constants.html#True), include these values.
- **zip_results** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `True` (https://docs.python.org/3/library/constants.html#True) and `include_obstime` is `True` (https://docs.python.org/3/library/constants.html#True), the return values will be a list of 2-tuples. If false, the return will be a single 2-tuple, where each element is a list.

Returns

values – Returns the list of values. See `include_obstime` and `zip_results` for modifications.

Return type

[value, . . .] or [(obstime, value), . . .] or ([obstime,. . .], [value, . . .])

Raises

requests.exceptions.HTTPError – Either a bad URL or non-existant mnemonic.

set_session()

Setup HTTP session

jwst.lib.engdb_lib Module

Engineering DB common library

Classes

EngDB_Value(obstime, value)	Create new instance of EngDB_Value(obstime, value)
EngdbABC([base_url])	Access the JWST Engineering Database

EngDB_Value

class jwst.lib.engdb_lib.**EngDB_Value**(*obstime*, *value*)

Bases: tuple (https://docs.python.org/3/library/stdtypes.html#tuple)

Create new instance of EngDB_Value(obstime, value)

Attributes Summary

obstime	Alias for field number 0
value	Alias for field number 1

Attributes Documentation

obstime

Alias for field number 0

value

Alias for field number 1

EngdbABC

class jwst.lib.engdb_lib.**EngdbABC**(*base_url=None*, ***service_kwargs*)

Bases: ABC (https://docs.python.org/3/library/abc.html#abc.ABC)

Access the JWST Engineering Database

This is the minimal API for the service definition. Concrete implementations may provide other parameters and attributes.

Parameters

- **base_url** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The base url for the engineering RESTful service
- **service_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Service-specific keyword arguments. Refer to the concrete implementations of EngdbABC.

Attributes Summary

base_url	The URL of the service in use
endtime	The endtime of the search
response	The `requests.Response` information
starttime	The start time of the search

Methods Summary

get_meta([mnemonic])	Get the mnemonics meta info
get_values(mnemonic, starttime, endtime[, ...])	Retrieve all results for a mnemonic in the requested time range.

Attributes Documentation

base_url

The URL of the service in use

endtime

The endtime of the search

response

The `requests.Response` information

starttime

The start time of the search

Methods Documentation

abstract get_meta(*mnemonic=''*, ***service_kwargs*)

Get the mnemonics meta info

Parameters

mnemonic (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The engineering mnemonic to retrieve

Returns

- **meta** (*object*) – The meta information. Type of return is dependent on the type of service
- **service_kwargs** (*dict*) – Service-specific keyword arguments. Refer to the concrete implementations of EngdbABC.

abstract get_values(*mnemonic*, *starttime*, *endtime*, *time_format=None*, *include_obstime=False*, *include_bracket_values=False*, *zip_results=True*)

Retrieve all results for a mnemonic in the requested time range.

Parameters

- **mnemonic** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The engineering mnemonic to retrieve
- **starttime** (str or `astropy.time.Time`) – The, inclusive, start time to retrieve from.

- **endtime** (str or `astropy.time.Time`) – The, inclusive, end time to retrieve from.
- **time_format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format of the input time used if the input times are strings. If None, a guess is made.
- **include_obstime** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `True` (https://docs.python.org/3/library/constants.html#True), the return values will include observation time as `astropy.time.Time`. See `zip_results` for further details.
- **include_bracket_values** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – The DB service, by default, returns the bracketing values outside of the requested time. If `True` (https://docs.python.org/3/library/constants.html#True), include these values.
- **zip_results** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `True` (https://docs.python.org/3/library/constants.html#True) and `include_obstime` is `True` (https://docs.python.org/3/library/constants.html#True), the return values will be a list of 2-tuples. If false, the return will be a single 2-tuple, where each element is a list.

Returns

values – Returns the list of values. See `include_obstime` and `zip_results` for modifications.

Return type

[value, . . .] or [(obstime, value), . . .] or ([obstime,. . .], [value, . . .])

Raises

requests.exceptions.HTTPError – Either a bad URL or non-existant mnemonic.

Telescope Pointing Utilities

jwst.lib.set_telescope_pointing Module

Set Telescope Pointing from Observatory Engineering Telemetry

Calculate and update the pointing-related and world coordinate system-related keywords. Given a time period, usually defined by an exposure, the engineering mnemonic database is queried for observatory orientation. The orientation defines the sky coordinates a particular point on the observatory is pointed to. Then, using a set of matrix transformations, the sky coordinates of the reference pixel of a desired aperture is calculated.

The transformations are defined by the Technical Reference JWST-STScI-003222, SM-12. This document has undergone a number of revisions. The current version implemented is based on an internal email version Rev. C, produced 2021-11.

There are a number of algorithms, or *methods*, that have been implemented. Most represent the historical refinement of the algorithm. Until the technical reference is finalized, all methods will remain in the code. The default, state-of-the art algorithm is represented by method `OPS_TR_202111`, implemented by *calc_transforms_ops_tr_202111*.

Interface

The primary usage is through the command line interface `set_telescope_pointing.py`. Operating on a list of JWST Level 1b exposures, this command updates the world coordinate system keywords with the values necessary to translate from aperture pixel to sky coordinates.

Access to the JWST Engineering Mnemonic database is required. See the *Engineering Database Interface* for more information.

Programmatically, the command line is implemented by the function *add_wcs*, which calls the basic function *calc_wcs*. The available methods are defined by *Methods*.

There are two data structures used to maintain the state of the transformation. *TransformParameters* contains the parameters needed to perform the transformations. *Transforms* contains the calculated transformation matrices.

Transformation Matrices

All the transformation matrices, as defined by *Transforms*, are Direction Cosine Matrices (DCM). A DCM contains the Euler rotation angles that represent the sky coordinates for a particular frame-of-reference. The initial DCM is provided through the engineering telemetry and represents where in the sky either the Fine Guidance Sensor (FGS) or star tracker is pointed to. Then, through a set of transformations, the DCM for the reference point of the target aperture is calculated.

Functions

add_wcs(filename[, allow_any_file, ...])	Add WCS information to a JWST DataModel.
calc_transforms(t_pars)	Calculate transforms which determine reference point celestial WCS
calc_transforms_ops_tr_202111(t_pars)	Calculate transforms in OPS using TR 2021-11
calc_wcs(t_pars)	Given observatory orientation and target aperture, calculate V1 and Reference Pixel sky coordinates
calc_wcs_over_time(obsstart, obsend, t_pars)	Calculate V1 and WCS over a time period
update_wcs(model[, default_pa_v3, ...])	Update WCS pointing information

add_wcs

jwst.lib.set_telescope_pointing.**add_wcs**(*filename*, *allow_any_file=False*, *force_level1bmodel=False*, *default_pa_v3=0.0*, *siaf_path=None*, *prd=None*, *engdb_url=None*, *fgsid=None*, *tolerance=60*, *allow_default=False*, *reduce_func=None*, *dry_run=False*, *save_transforms=None*, ***transform_kwargs*)

Add WCS information to a JWST DataModel.

Telescope orientation is attempted to be obtained from the engineering database. Failing that, a default pointing is used based on proposal target.

The file is updated in-place.

Parameters

- **filename** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to a data file.
- **allow_any_file** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Attempt to add the WCS information to any type of file. The default, `False` (https://docs.python.org/3/library/constants.html#False), only allows modifications of files that contain known datamodels of `Level1bmodel`, `ImageModel`, or `CubeModel`.
- **force_level1bmodel** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If not `allow_any_file`, and the input file model is unknown, open the input file as a Level1bModel regardless.
- **default_pa_v3** (*float* (https://docs.python.org/3/library/functions.html#float)) – The V3 position angle to use if the pointing information is not found.

- **siaf_path** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or file-like object or None*) – The path to the SIAF database. See `SiafDb` for more information.
- **prd** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The PRD version from the `pysiaf` to use. `siaf_path` overrides this value.
- **engdb_url** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – URL of the engineering telemetry database REST interface.
- **fgsid** (*int* (https://docs.python.org/3/library/functions.html#int) *or None*) – When in COARSE mode, the FGS to use as the guider reference. If None, use what is provided in telemetry.
- **tolerance** (*int* (https://docs.python.org/3/library/functions.html#int)) – If no telemetry can be found during the observation, the time, in seconds, beyond the observation time to search for telemetry.
- **allow_default** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If telemetry cannot be determine, use existing information in the observation's header.
- **reduce_func** (*func or None*) – Reduction function to use on values.
- **dry_run** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Do not write out the modified file.
- **save_transforms** (*Path-like or None*) – File to save the calculated transforms to.
- **transform_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Keyword arguments used by matrix calculation routines.

Notes

This function adds absolute pointing information to the JWST datamodels provided. By default, only Stage 1 and Stage 2a exposures are allowed to be updated. These have the suffixes of "uncal", "rate", and "rateints" representing datamodels Level1bModel, ImageModel, and CubeModel. Any higher level product, from Stage 2b and beyond, that has had the `assign_wcs` step applied, have improved WCS information. Running this task on such files will potentially corrupt the WCS.

It starts by populating the headers with values from the SIAF database. It adds the following keywords to all files:

V2_REF (arcseconds) V3_REF (arcseconds) VPARITY (+1 or -1) V3I_YANG (decimal degrees)

The keywords computed and added to all files are:

RA_V1 DEC_V1 PA_V3 RA_REF DEC_REF ROLL_REF S_REGION

In addition the following keywords are computed and added to IMAGING_MODES only:

CRVAL1 CRVAL2 PC1_1 PC1_2 PC2_1 PC2_2

It does not currently place the new keywords in any particular location in the header other than what is required by the standard.

calc_transforms

jwst.lib.set_telescope_pointing.**calc_transforms**(*t_pars:* TransformParameters)

Calculate transforms which determine reference point celestial WCS

This implements Eq. 3 from Technical Report JWST-STScI-003222, SM-12. Rev. C, 2021-11 From Section 3:

The Direction Cosine Matrix (DCM) that provides the transformation of a unit pointing vector defined in inertial frame (ECI J2000) coordinates to a unit vector defined in the science aperture Ideal frame coordinates is defined as [follows.]

Parameters
t_pars (TransformParameters) – The transformation parameters. Parameters are updated during processing.

Returns
transforms – The list of coordinate matrix transformations

Return type
Transforms

calc_transforms_ops_tr_202111

jwst.lib.set_telescope_pointing.**calc_transforms_ops_tr_202111**(*t_pars:* TransformParameters)

Calculate transforms in OPS using TR 2021-11

This implements the ECI-to-SIAF transformation from Technical Report JWST-STScI-003222, SM-12, Rev. C, 2021-11 The actual implementation depends on the guide star mode, represented by the header keyword PCS_MODE. For COARSE or NONE, the method COARSE is used. For TRACK or FINEGUIDE, the method TRACK is used.

Parameters
t_pars (TransformParameters) – The transformation parameters. Parameters are updated during processing.

Returns
transforms – The list of coordinate matrix transformations

Return type
Transforms

calc_wcs

jwst.lib.set_telescope_pointing.**calc_wcs**(*t_pars:* TransformParameters)

Given observatory orientation and target aperture, calculate V1 and Reference Pixel sky coordinates

Parameters
t_pars (*TransformParameters*) – The transformation parameters. Parameters are updated during processing.

Returns
wcsinfo, vinfo, transforms – A 3-tuple is returned with the WCS pointing for the aperture and the V1 axis, and the transformation matrices.

Return type
WCSRef, *WCSRef*, *Transforms*

calc_wcs_over_time

jwst.lib.set_telescope_pointing.**calc_wcs_over_time**(*obsstart*, *obsend*, *t_pars:* TransformParameters)

Calculate V1 and WCS over a time period

Parameters

- **obsstart** (*float* (https://docs.python.org/3/library/functions.html#float)) – MJD observation start/end times
- **obsend** (*float* (https://docs.python.org/3/library/functions.html#float)) – MJD observation start/end times
- **t_pars** (*TransformParameters*) – The transformation parameters. Parameters are updated during processing.

Returns

obstimes, wcsinfos, vinfos – A 3-tuple is returned with the WCS pointings for the aperture and the V1 axis

Return type

[astropy.time.Time[,...]], [*WCSRef*[,...]], [*WCSRef*[,...]]

update_wcs

jwst.lib.set_telescope_pointing.**update_wcs**(*model*, *default_pa_v3=0.0*, *default_roll_ref=0.0*, *siaf_path=None*, *prd=None*, *engdb_url=None*, *fgsid=None*, *tolerance=60*, *allow_default=False*, *reduce_func=None*, ***transform_kwargs*)

Update WCS pointing information

Given a jwst.datamodels.JwstDataModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel.h determine the simple WCS parameters from the SIAF keywords in the model and the engineering parameters that contain information about the telescope pointing.

It presumes all the accessed keywords are present (see first block).

Parameters

- **model** (JwstDataModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel.html#jv – The model to update.
- **default_roll_ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – If pointing information cannot be retrieved, use this as the roll ref angle.
- **siaf_path** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or Path-like object*) – The path to the SIAF database. See SiafDb for more information.
- **prd** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The PRD version from the pysiaf to use. siaf_path overrides this value.
- **engdb_url** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None*) – URL of the engineering telemetry database REST interface.
- **fgsid** (*int* (https://docs.python.org/3/library/functions.html#int) *or None*) – When in COARSE mode, the FGS to use as the guider reference. If None, use what is provided in telemetry.

- **tolerance** (*int* (https://docs.python.org/3/library/functions.html#int)) – If no telemetry can be found during the observation, the time, in seconds, beyond the observation time to search for telemetry.
- **allow_default** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If telemetry cannot be determine, use existing information in the observation's header.
- **reduce_func** (*func or None*) – Reduction function to use on values.
- **transform_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Keyword arguments used by matrix calculation routines.

Returns

t_pars, transforms – The parameters and transforms calculated. May be None for either if telemetry calculations were not performed. In particular, FGS GUIDER data does not need `transforms`.

Return type

TransformParameters, *Transforms*

Classes

Methods(value[, names, module, qualname, ...])	Available methods to calculate V1 and aperture WCS information
TransformParameters([allow_default, ...])	Parameters required the calculations
Transforms([m_eci2fgs1, m_eci2gs, m_eci2j, ...])	The matrices used in calculation of the M_eci2siaf transformation
WCSRef(ra, dec, pa)	Create new instance of WCSRef(ra, dec, pa)

Methods

class jwst.lib.set_telescope_pointing.**Methods**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum (https://docs.python.org/3/library/enum.html#enum.Enum)

Available methods to calculate V1 and aperture WCS information

Current state-of-art is OPS_TR_202111. This method chooses either COARSE_TR_202111 or TRACK_TR_202111 depending on the guidance mode, as specified by header keyword PCS_MODE.

Attributes Summary

COARSE	Default algorithm under PCS_MODE COARSE.
COARSE_TR_202111	COARSE tracking mode algorithm, TR version 2021-11.
OPS	Default algorithm for use by Operations.
OPS_TR_202111	Method to use in OPS to use TR version 2021-11
TRACK	Default algorithm under PCS_MODE TRACK/FINEGUIDE/MOVING.
TRACK_TR_202111	TRACK and FINEGUIDE mode algorithm, TR version 2021-11
calc_func	Function associated with the method
default	Algorithm to use by default.
func	Function associated with the method
mnemonics	

Attributes Documentation

COARSE = 'coarse_tr_202111'

Default algorithm under PCS_MODE COARSE.

COARSE_TR_202111 = 'coarse_tr_202111'

COARSE tracking mode algorithm, TR version 2021-11.

OPS = 'ops_tr_202111'

Default algorithm for use by Operations.

OPS_TR_202111 = 'ops_tr_202111'

Method to use in OPS to use TR version 2021-11

TRACK = 'track_tr_202111'

Default algorithm under PCS_MODE TRACK/FINEGUIDE/MOVING.

TRACK_TR_202111 = 'track_tr_202111'

TRACK and FINEGUIDE mode algorithm, TR version 2021-11

calc_func

Function associated with the method

default = 'ops_tr_202111'

Algorithm to use by default. Used by Operations.

func

Function associated with the method

mnemonics

TransformParameters

class jwst.lib.set_telescope_pointing.**TransformParameters**(*allow_default: bool (https://docs.python.org/3/library/functions.html#bool) = False*, *default_pa_v3: float (https://docs.python.org/3/library/functions.html#float) = 0.0*, *detector: str (https://docs.python.org/3/library/stdtypes.html#str) = None*, *dry_run: bool (https://docs.python.org/3/library/functions.html#bool) = False*, *engdb_url: str (https://docs.python.org/3/library/stdtypes.html#str) = None*, *exp_type: str (https://docs.python.org/3/library/stdtypes.html#str) = None*, *fgsid: int (https://docs.python.org/3/library/functions.html#int) = None*, *fsmcorr_version: str (https://docs.python.org/3/library/stdtypes.html#str) = 'latest'*, *fsmcorr_units: str (https://docs.python.org/3/library/stdtypes.html#str) = 'arcsec'*, *guide_star_wcs: WCSRef = (None, None, None)*, *j2fgs_transpose: bool (https://docs.python.org/3/library/functions.html#bool) = True*, *jwst_velocity: array = None*, *method: Methods = Methods.OPS_TR_202111*, *obsend: float (https://docs.python.org/3/library/functions.html#float) = None*, *obsstart: float (https://docs.python.org/3/library/functions.html#float) = None*, *override_transforms: Transforms = None*, *pcs_mode: str (https://docs.python.org/3/library/stdtypes.html#str) = None*, *pointing: Pointing = None*, *reduce_func: Callable (https://docs.python.org/3/library/typing.html#typing.Calla = None*, *siaf: SIAF = None*, *siaf_db: SiafDb = None*, *tolerance: float (https://docs.python.org/3/library/functions.html#float) = 60.0*, *useafter: str (https://docs.python.org/3/library/stdtypes.html#str) = None*, *v3pa_at_gs: float (https://docs.python.org/3/library/functions.html#float) = None*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Parameters required the calculations

Attributes Summary

allow_default	If telemetry cannot be determined, use existing information in the observation's header.
default_pa_v3	The V3 position angle to use if the pointing information is not found.
detector	Detector in use.
dry_run	Do not write out the modified file.
engdb_url	URL of the engineering telemetry database REST interface.
exp_type	Exposure type
fgsid	FGS to use as the guiding FGS.
fsmcorr_units	Units of the FSM correction values.
fsmcorr_version	The version of the FSM correction calculation to use.
guide_star_wcs	Guide star WCS info, typically from the input model.
j2fgs_transpose	Transpose the `j2fgs1` matrix.
jwst_velocity	The [DX, DY, DZ] barycentri velocity vector
method	The method, or algorithm, to use in calculating the transform.
obsend	Observation end time
obsstart	Observation start time
override_transforms	If set, matrices that should be used instead of the calculated one.
pcs_mode	The tracking mode in use.
pointing	The observatory orientation, represented by the ECI quaternion, and other engineering mnemonics
reduce_func	Reduction function to use on values.
siaf	The SIAF information for the input model
siaf_db	The SIAF database
tolerance	If no telemetry can be found during the observation, the time, in seconds, beyond the observation time to search for telemetry.
useafter	The date of observation (`jwst.datamodels.JwstDataModel.meta.date`)
v3pa_at_gs	V3 position angle at Guide Star (`jwst.datamodels.JwstDataModel.meta.guide_star.gs_v3_pa_science`)

Methods Summary

as_reprdict()	Return a dict where all values are REPR of their values
update_pointing()	Update pointing information

Attributes Documentation

allow_default: *bool* (https://docs.python.org/3/library/functions.html#bool) = False
If telemetry cannot be determined, use existing information in the observation's header.

default_pa_v3: *float* (https://docs.python.org/3/library/functions.html#float) = 0.0
The V3 position angle to use if the pointing information is not found.

detector: *str* (https://docs.python.org/3/library/stdtypes.html#str) = None
Detector in use.

dry_run: *bool* (https://docs.python.org/3/library/functions.html#bool) = False
Do not write out the modified file.

engdb_url: *str* (https://docs.python.org/3/library/stdtypes.html#str) = None
URL of the engineering telemetry database REST interface.

exp_type: *str* (https://docs.python.org/3/library/stdtypes.html#str) = None
Exposure type

fgsid: *int* (https://docs.python.org/3/library/functions.html#int) = None
FGS to use as the guiding FGS. If None, will be set to what telemetry provides.

fsmcorr_units: *str* (https://docs.python.org/3/library/stdtypes.html#str) = 'arcsec'
Units of the FSM correction values. Default is 'arcsec'. See `calc_sifov_fsm_delta_matrix`

fsmcorr_version: *str* (https://docs.python.org/3/library/stdtypes.html#str) = 'latest'
The version of the FSM correction calculation to use. See `calc_sifov_fsm_delta_matrix`

guide_star_wcs: *WCSRef* = (None, None, None)
Guide star WCS info, typically from the input model.

j2fgs_transpose: *bool* (https://docs.python.org/3/library/functions.html#bool) = True
Transpose the `j2fgs1` matrix.

jwst_velocity: *array* = None
The [DX, DY, DZ] barycentri velocity vector

method: *Methods* = 'ops_tr_202111'
The method, or algorithm, to use in calculating the transform. If not specified, the default method is used.

obsend: *float* (https://docs.python.org/3/library/functions.html#float) = None
Observation end time

obsstart: *float* (https://docs.python.org/3/library/functions.html#float) = None
Observation start time

override_transforms: *Transforms* = None
If set, matrices that should be used instead of the calculated one.

pcs_mode: *str* (https://docs.python.org/3/library/stdtypes.html#str) = None
The tracking mode in use.

pointing: *Pointing* = None
The observatory orientation, represented by the ECI quaternion, and other engineering mnemonics

reduce_func: Callable (https://docs.python.org/3/library/typing.html#typing.Callable) = None

Reduction function to use on values.

siaf: SIAF = None

The SIAF information for the input model

siaf_db: SiafDb = None

The SIAF database

tolerance: float (https://docs.python.org/3/library/functions.html#float) = 60.0

If no telemetry can be found during the observation, the time, in seconds, beyond the observation time to search for telemetry.

useafter: str (https://docs.python.org/3/library/stdtypes.html#str) = None

The date of observation (`jwst.datamodels.JwstDataModel.meta.date`)

v3pa_at_gs: float (https://docs.python.org/3/library/functions.html#float) = None

V3 position angle at Guide Star (`jwst.datamodels.JwstDataModel.meta.guide_star.gs_v3_pa_science`)

Methods Documentation

as_reprdict()

Return a dict where all values are REPR of their values

update_pointing()

Update pointing information

Transforms

class jwst.lib.set_telescope_pointing.**Transforms**(*m_eci2fgs1: array = None*, *m_eci2gs: array = None*, *m_eci2j: array = None*, *m_eci2siaf: array = None*, *m_eci2sifov: array = None*, *m_eci2v: array = None*, *m_fgsx2gs: array = None*, *m_fgs12sifov: array = None*, *m_gs2gsapp: array = None*, *m_j2fgs1: array = None*, *m_sifov_fsm_delta: array = None*, *m_sifov2v: array = None*, *m_v2siaf: array = None*, *override: object (https://docs.python.org/3/library/functions.html#object) = None*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

The matrices used in calculation of the M_eci2siaf transformation

Attributes Summary

m_eci2fgs1	ECI to FGS1
m_eci2gs	ECI to Guide Star
m_eci2j	ECI to J-Frame
m_eci2siaf	ECI to SIAF
m_eci2sifov	ECI to SIFOV
m_eci2v	ECI to V
m_fgs12sifov	FGS1 to SIFOV
m_fgsx2gs	FGSX to Guide Stars transformation
m_gs2gsapp	Velocity aberration
m_j2fgs1	J-Frame to FGS1
m_sifov2v	SIFOV to V1
m_sifov_fsm_delta	FSM correction
m_v2siaf	V to SIAF
override	Override values.

Methods Summary

from_asdf(asdf_file)	Create Transforms from AsdfFile
to_asdf()	Serialize to AsdfFile
write_to_asdf(path)	Serialize to a file path

Attributes Documentation

m_eci2fgs1: array = None

ECI to FGS1

m_eci2gs: array = None

ECI to Guide Star

m_eci2j: array = None

ECI to J-Frame

m_eci2siaf: array = None

ECI to SIAF

m_eci2sifov: array = None

ECI to SIFOV

m_eci2v: array = None

ECI to V

m_fgs12sifov: array = None

FGS1 to SIFOV

m_fgsx2gs: array = None

FGSX to Guide Stars transformation

m_gs2gsapp: array = None

Velocity aberration

m_j2fgs1: array = None
J-Frame to FGS1

m_sifov2v: array = None
SIFOV to V1

m_sifov_fsm_delta: array = None
FSM correction

m_v2siaf: array = None
V to SIAF

override: object (https://docs.python.org/3/library/functions.html#object) = None
Override values. Either another Transforms or dict-like object

Methods Documentation

classmethod from_asdf(*asdf_file*)
Create Transforms from AsdfFile

Parameters
asdf_file (Stream-like or `asdf.AsdfFile`) – The asdf to create from.

Returns
transforms – The Transforms instance.

Return type
Transforms

to_asdf()
Serialize to AsdfFile

Returns
asdf_file – The ASDF serialization.

Return type
asdf.AsdfFile

Notes

The *override* transforms are not serialized, since the values of this transform automatically represent what is in the override.

write_to_asdf(*path*)
Serialize to a file path

Parameters
path (*Stream-like*) –

WCSRef

class jwst.lib.set_telescope_pointing.**WCSRef**(*ra*, *dec*, *pa*)

Bases: tuple (https://docs.python.org/3/library/stdtypes.html#tuple)

Create new instance of WCSRef(ra, dec, pa)

Attributes Summary

dec	Alias for field number 1
pa	Alias for field number 2
ra	Alias for field number 0

Attributes Documentation

dec

Alias for field number 1

pa

Alias for field number 2

ra

Alias for field number 0

jwst.lib.v1_calculate Module

V1 Calculation based on time and engineering database info

Functions

v1_calculate_from_models(sources[, siaf_path])	Calculate V1 over the time period for the given models
v1_calculate_over_time(obsstart, obsend[, ...])	Calculate V1 over the given time period

v1_calculate_from_models

jwst.lib.v1_calculate.**v1_calculate_from_models**(*sources*, *siaf_path=None*, ***calc_wcs_from_time_kwargs*)

Calculate V1 over the time period for the given models

Returns a table of V1 pointings for all input models. The table has the following columns:

- source (jwst.datamodels.JwstDataModel): The model
- obstime (astropy.time.Time): The observation time
- v1 (float, float, float): 3-tuple or ra, dec, and position angle

Parameters

- **sources** (*[File-like or jwst.datamodels.Datamodel[...]]*) – The datamodels to get timings other header parameters from.
- **siaf_path** (*None or file-like*) – The path to the SIAF database. If none, the default used by the `pysiaf` package is used. See `SiafDb` for more information.
- **calc_wcs_from_time_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Keyword arguments to pass to `calc_wcs_from_time`

Returns
v1_table – Table of V1 pointing

Return type
astropy.table.Table

v1_calculate_over_time

`jwst.lib.v1_calculate.`**v1_calculate_over_time**(*obsstart*, *obsend*, *siaf_path=None*, ***calc_wcs_from_time_kwargs*)

Calculate V1 over the given time period

Returns a table of all V1 pointings that can be retrieved from the engineering database that exist between, inclusively, the start and end times.

The table has the following columns:

- source (str): The string "time range"
- obstime (astropy.time.Time): The observation time
- v1 (float, float, float): 3-tuple or ra, dec, and position angle

Parameters

- **obsstart** (*float* (https://docs.python.org/3/library/functions.html#float)) – The MJD start and end time to search for pointings.
- **obsend** (*float* (https://docs.python.org/3/library/functions.html#float)) – The MJD start and end time to search for pointings.
- **siaf_path** (*None or file-like*) – The path to the SIAF database. If none, the default used by the `pysiaf` package is used. See `SiafDb` for more information.
- **calc_wcs_from_time_kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Keyword arguments to pass to `calc_wcs_from_time`

Returns
v1_table – Table of V1 pointing

Return type
astropy.table.Table

Commands

Available commands are as follows. Use the -h option for more details.

set_telescope_pointing.py
Update basic WCS information in JWST exposures from the engineering database.

pointing_summary
Summarize various pointing information in a table.

v1_calculate
Calculate V1 over a time period.

15.1.33 Last Frame Correction

Description

Class
jwst.lastframe.LastFrameStep

Alias
lastframe

The last frame correction step is only applied to MIRI data and flags the final group in each integration as bad (the "DO_NOT_USE" bit is set in the GROUPDQ flag array), but only if the total number of groups in each integration is greater than 2. This results in the data contained in the last group being excluded from subsequent steps, such as jump detection and ramp fitting. No flags are added if NGROUPS <= 2, because doing so would leave too few good groups to work with in later steps.

Only the GROUPDQ array is modified. The SCI, ERR, and PIXELDQ arrays are unchanged.

Step Arguments

The last frame correction has no step-specific arguments.

Reference File

This step does not use any reference file.

jwst.lastframe Package

Classes

LastFrameStep([name, parent, config_file, ...])	LastFrameStep: This is a MIRI specific task.

LastFrameStep

class jwst.lastframe.**LastFrameStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

LastFrameStep: This is a MIRI specific task. If the number of groups is greater than 2, the GROUP data quality flags for the final group will be set to DO_NOT_USE.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'lastframe'

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.34 Linearity Correction

Description

Class
jwst.linearity.LinearityStep

Alias
linearity

Assumptions

It is assumed that the input science exposure data for near-IR instruments have had the *superbias* subtraction step applied, therefore the correction coefficients stored in the linearity reference files for those instruments must have been derived from data that has also been bias subtracted. MIRI data, on the other hand, do not receive bias subtraction (see *calwebb_detector1*) and hence the linearity correction is derived from data that has not been bias subtracted.

It is also assumed that the saturation step has already been applied to the input data, so that saturation flags are set in the GROUPDQ array of the input science data.

Algorithm

The algorithm for this step is called from the external package `stcal`, an STScI effort to unify common calibration processing algorithms for use by multiple observatories.

The linearity step applies the "classic" linearity correction adapted from the HST WFC3/IR linearity correction routine, correcting science data values for detector non-linearity. The correction is applied pixel-by-pixel, group-by-group, integration-by-integration within a science exposure.

The correction is represented by an nth-order polynomial for each pixel in the detector, with n+1 arrays of coefficients read from the linearity reference file.

The algorithm for correcting the observed pixel value in each group of an integration is currently of the form:

$$F_{\rm c} = c_0 + c_1 F + c_2 F^2 + c_3 F^3 + ... + c_n F^n$$

where F is the observed counts (in DN), c_n are the polynomial coefficients, and $F_{\rm c}$ is the corrected counts. There is no limit to the order of the polynomial correction; all coefficients contained in the reference file will be applied.

Upon successful completion of the linearity correction the S_LINEAR keyword is set to "COMPLETE".

Special Handling

1. Pixels having at least one correction coefficient equal to NaN will not have the linearity correction applied and the DQ flag "NO_LIN_CORR" is added to the science exposure PIXELDQ array.
2. Pixels that have the "NO_LIN_CORR" flag set in the DQ array of the linearity reference file will not have the correction applied and the "NO_LIN_CORR" flag is added to the science exposure PIXELDQ array.
3. Pixel values that have the "SATURATED" flag set in a particular group of the science exposure GROUPDQ array will not have the linearity correction applied to that group. Any groups for that pixel that are not flagged as saturated will be corrected.

The ERR array of the input science exposure is not modified.

The flags from the linearity reference file DQ array are propagated into the PIXELDQ array of the science exposure using a bitwise OR operation.

NIRCam Frame 0

If the NIRCam "Frame 0" data are included in the input, the linearity correction is applied to each integration's frame zero image in the same way as it's applied to the normal science data cube. The corrected frame zero data are returned as part of the overall datamodel being processed.

Subarrays

This step handles input science exposures that were taken in subarray modes in a flexible way. If the reference data arrays are the same size as the science data, they will be applied directly. If there is a mismatch, the routine will extract a matching subarray from the reference file data arrays and apply them to the science data. Hence full-frame reference files can be used for both full-frame and subarray science exposures, or subarray-dependent reference files can be provided if desired.

Arguments

The linearity correction has no step-specific arguments.

Reference File Types

The `linearity` step uses a LINEARITY reference file.

LINEARITY Reference File

REFTYPE
LINEARITY

Data model
`LinearityModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.LinearityModel.html#jwst.datamodels.

The LINEARITY reference file contains pixel-by-pixel polynomial correction coefficients.

Reference Selection Keywords for LINEARITY

CRDS selects appropriate LINEARITY references based on the following keywords. LINEARITY is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, SUBARRAY, BAND, FILTER, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for LINEARITY

In addition to the standard reference file keywords listed above, the following keywords are *required* in LINEARITY reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for LINEARITY*):

Keyword	Data Model Name	Instruments
DETECTOR	model.meta.instrument.detector	All
SUBARRAY	model.meta.subarray.name	All
FILTER	model.meta.instrument.filter	MIRI only
BAND	model.meta.instrument.band	MIRI only

Reference File Format

LINEARITY reference files are FITS format, with 2 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
COEFFS	IMAGE	3	ncols x nrows x ncoeffs	float
DQ	IMAGE	2	ncols x nrows	integer
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

Each plane of the COEFFS data cube contains the pixel-by-pixel coefficients for the associated order of the polynomial. There can be any number of planes to accommodate a polynomial of any order.

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

jwst.linearity Package

Classes

LinearityStep([name, parent, config_file, ...])	LinearityStep: This step performs a correction for non-linear detector response, using the "classic" polynomial method.

LinearityStep

class jwst.linearity.**LinearityStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

LinearityStep: This step performs a correction for non-linear detector response, using the "classic" polynomial method.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.

- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'linearity'

reference_file_types = ['linearity']

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.35 Master Background Subtraction

Description

Classes

jwst.master_background.MasterBackgroundStep, *jwst.master_background.MasterBackgroundMosStep*

Aliases

master_background, master_background_mos

Master background subtraction is one form of background subtraction available for spectroscopic data. See *Background Subtraction* for an overview of all the available methods and where they occur within the various stages of the calibration pipeline.

The master background subtraction step subtracts background signal from 2-D spectroscopic data using a 1-D master background spectrum. The 1-D master background spectrum is created from one or more input exposures, or can alternatively be supplied by the user. The 1-D background spectrum - surface brightness versus wavelength - is projected into the 2-D space of source data based on the wavelength of each pixel in the 2-D data. The resulting 2-D background signal is then subtracted directly from the 2-D source data.

Logic built into the step checks to see if the exposure-based *background* subtraction step in the *calwebb_spec2* pipeline has already been performed on the input images, based on the value of the S_BKDSUB keyword. If S_BKGSUB is set to "COMPLETE", the master background step is skipped. If the *calwebb_spec2* background step was not applied, the master background step will proceed. The user can override this logic, if desired, by setting the step argument `--force_subtract` to `True`, in which case master background subtraction will be applied regardless of the value of S_BKDSUB (see *Step Arguments*).

Upon successful completion of the step, the S_MSBSUB keyword is set to "COMPLETE" in the output product. The background-subtracted results are returned as a new data model, leaving the input model unchanged.

Note: The application of master background subtraction to NIRSpec Fixed-Slit, IFU, and MOS observations requires special handling, due to unique types of calibrations that are applied to these modes. NIRSpec MOS mode requires even more special handling than NIRSpec Fixed-Slit and IFU. The next several sections pertain primarily to MIRI MRS and LRS Fixed-Slit, and in a general way to NIRSpec Fixed-Slit and IFU modes. Details regarding all NIRSpec modes are given later in *NIRSpec Master Background Subtraction*.

Inputs

The primary driver of the master background step is usually a `spec3` type Association (ASN) file or a `ModelContainer` data model populated from a `spec3` ASN file. This is the same ASN file used as input to the *calwebb_spec3* pipeline, which defines a stage 3 combined product and its input members. The list of input members includes both "science" and "background" exposure types. The master background subtraction step uses the input members designated with `"exptype": "background"` to create the master background spectrum (see *example_asn1*). These need to be *x1d* products created from individual exposures at the end of the *calwebb_spec2* pipeline, containing spectra of background regions. The master background signal will be subtracted from all input members designated as `"exptype": "science"` in the ASN, resulting in a new version of each science input. These inputs need to be *cal* products created from individual exposures by the *calwebb_spec2* pipeline.

There are two main observing scenarios that are supported by this step: nodded exposures of point sources and off-source background exposures of extended targets. A third type of operation is performed for NIRSpec MOS observations that include background slits. The details for each mode are explained below.

Nodded Point Sources

If an observation uses a nodding type dither pattern to move a small or point-like source within the field-of-view, it is assumed that part of the field-of-view in each exposure is also suitable for measuring background. Exposures of this type are identified by the pipeline based on their "PATTTYPE" (primary dither pattern type) keyword value. The value will either contain the substring "NOD" somewhere within the name (e.g. "2-POINT-NOD" or "ALONG-SLIT-NOD"), or will be set to "POINT-SOURCE" (for MIRI MRS). The *calwebb_spec2 srctype* step recognizes these PATTTYPE values and sets the source type to "POINT."

This in turn causes the *extract_1d* step at the end of *calwebb_spec2* to extract spectra for both source and background regions. For IFU exposures the background region is typically an annulus that is concentric with a circular source region. For slit-like modes, one or more background regions can be defined in the *extract1d* reference file, flanking the central source region. In both cases, the extraction regions are centered within the image/cube at the RA/Dec of the target. Hence for nodded exposures, the location of the extraction regions follows the movement of the source in each exposure. The extracted data from the source region are stored in the "FLUX" and "SURF_BRIGHT" (surface brightness) columns of the *x1d* product, while the background extraction is stored in the "BACKGROUND" column. The `master_background` step uses the data from the "BACKGROUND" column of each background *x1d* product to create the 1-D master background spectrum.

Below is an example ASN file for a simple 2-point nodded observation consisting of two exposures.

```
{
    "asn_type": "spec3",
    "asn_rule": "candidate_Asn_IFU",
    "program": "00626",
    "asn_id": "c1003",
    "target": "t001",
    "asn_pool": "jw00626_20190128T194403_pool",
    "products": [
        {"name": "jw00626-c1003_t001_nrs",
            "members": [
                {"expname": "jw00626009001_02101_00001_nrs1_cal.fits",
                  "exptype": "science",
                  "asn_candidate": "('c1003', 'background')"
                },
                {"expname": "jw00626009001_02102_00001_nrs1_cal.fits",
                 "exptype": "science",
                 "asn_candidate": "('c1003', 'background')"
                },
                {"expname": "jw00626009001_02101_00001_nrs1_x1d.fits",
                 "exptype": "background",
                 "asn_candidate": "('c1003', 'background')"
                },
                {"expname": "jw00626009001_02102_00001_nrs1_x1d.fits",
                 "exptype": "background",
                 "asn_candidate": "('c1003', 'background')"
                }
            ]
        }
    ]
}
```

As you can see, the same two exposures are defined as being both "science" and "background" members, because they both contain the target of interest and a region of background. The "science" members, which are the *cal* products created by the *calwebb_spec2* pipeline, are the data files that will have the master background subtraction applied,

while the "background" members are the *x1d* spectral products from which the master background spectrum will be created. The combined master background spectrum will be subtracted from each of the two science exposures.

Extended Source with Dedicated Background Exposures

Observations of extended sources must obtain exposures of a separate background target/field in order to measure the background. Exposures of a background target are identified by the keyword "BKGDTARG" set to `True` (https://docs.python.org/3/library/constants.html#True) in the header. During *calwebb_spec2* processing, the *srctype* step recognizes these and sets their source type to "EXTENDED", because all dedicated background exposures are to be processed as extended sources.

This in turn causes the *extract_1d* step at the end of *calwebb_spec2* to extract a spectrum in extended source mode, which uses the entire field-of-view (whether it be a slit image or an IFU cube) as the extraction region. The `master_background` step recognizes which type of background exposure it's working with and uses the appropriate data from the *x1d* product to construct the master background spectrum.

Below is an example ASN file for an extended source observation that includes background target exposures, using a 2-point dither for both the science and background targets.

```
{
    "asn_type": "spec3",
    "asn_rule": "candidate_Asn_IFU",
    "program": "00626",
    "asn_id": "c1004",
    "target": "t002",
    "asn_pool": "jw00626_20190128T194403_pool",
    "products": [
        {"name": "jw00626-c1004_t002_nrs",
            "members": [
                {"expname": "jw00626009001_02101_00001_nrs1_cal.fits",
                  "exptype": "science",
                  "asn_candidate": "('c1004', 'background')"
                },
                {"expname": "jw00626009001_02102_00001_nrs1_cal.fits",
                 "exptype": "science",
                 "asn_candidate": "('c1004', 'background')"
                },
                {"expname": "jw00626009001_02103_00001_nrs1_x1d.fits",
                 "exptype": "background",
                 "asn_candidate": "('c1004', 'background')"
                },
                {"expname": "jw00626009001_02104_00001_nrs1_x1d.fits",
                 "exptype": "background",
                 "asn_candidate": "('c1004', 'background')"
                }
            ]
        }
    ]
}
```

In this example there are two exposures of the science target, labeled as "science" members, and two exposures of the background target, labeled as "background" members. As before, the science members use *cal* products as input and the background members use *x1d* products as input. The master background step will first combine the data from the two background members into a master background spectrum and then subtract it from each of the two science

exposures.

Creating the 1-D Master Background Spectrum

The 1-D master background spectrum is created by combining data contained in the *x1d* products listed in the input ASN as `"exptype": "background"` members. As noted above, the background members can be exposures of dedicated background targets or can be a collection of exposures of a point-like source observed in a nod pattern.

When all of the input background spectra have been collected, they are combined using the *combine_1d* step to produce the 1-D master background spectrum. See the *combine_1d* step for more details on the processes used to create the combined spectrum.

Subtracting the Master Background

The 1-D master background spectrum is interpolated by wavelength at each pixel of a 2-D source spectrum and subtracted from it. The source data instances can be, for example, a set of NIRSpec or MIRI IFU exposures, a set of NIRSpec fixed-slit 2-D extractions, or a set of nodded MIRI LRS fixed-slit exposures. The subtraction is performed on all data instances within all input science exposures. For example, if there are 3 NIRSpec fixed-slit exposures, each containing data from multiple slits, the subtraction is applied one-by-one to all slit instances in all exposures. For each data instance to be subtracted the following steps are performed:

1. Compute a 2-D wavelength grid corresponding to the 2-D source data. For some observing modes, such as NIRSpec MOS and fixed-slit, a 2-D wavelength array is already computed and attached to the data in the *calwebb_spec2* pipeline *extract_2d* step. If such a wavelength array is present, it is used. For modes that don't have a 2-D wavelength array contained in the data, it is computed on the fly using the WCS object for each source data instance.
2. Compute the background signal at each pixel in the 2-D wavelength grid by interpolating within the 1-D master background spectrum as a function of wavelength. Pixels in the 2-D source data with an undefined wavelength (e.g. wavelength array value of NaN) or a wavelength that is beyond the limits of the master background spectrum receive special handling. The interpolated background value is set to zero and a DQ flag of "DO_NOT_USE" is set.
3. Subtract the resulting 2-D background image from the 2-D source data. DQ values from the 2-D background image are propagated into the DQ array of the subtracted science data.

NIRSpec Master Background Corrections

The master background subtraction methods and processing flow for NIRSpec Fixed-Slit and IFU modes is largely the same as what's outlined above, with some additional operations that need to be applied to accommodate some of the unique calibrations applied to NIRSpec data. NIRSpec MOS mode requires even more special handling. This is due to two primary effects of NIRSpec calibration:

1. Point sources in MOS and Fixed-Slit mode receive wavelength offset corrections if the source is not centered (along the dispersion direction) within the slit. Hence the wavelength grid assigned to each 2-D slit cutout can be shifted slightly relative to the wavelengths of the background signal contained in the same cutout. And because the flat-field, pathloss, and photom corrections/calibrations are wavelength-dependent, the pixel-level calibrations for the source signal are slightly different than the background.
2. Point sources and uniform sources receive different pathloss and bar shadow corrections (in fact point sources don't receive any bar shadow correction). So the background signal contained within a calibrated point source cutout has received a different pathloss correction and hasn't received any bar shadow correction. Meanwhile, the master background is created from data that had corrections for a uniform source applied to it and hence there's a mismatch relative to the point source data.

The 2-D background that's initially created from the 1-D master background is essentially a perfectly calibrated background signal. However, due to the effects mentioned above, the actual background signal contained within a calibrated point source slit (or IFU image) is not perfect (e.g. it still has the bar shadow effects in it). So all of these effects need to be accounted for in the computed 2-D background before subtracting from the source data.

NIRSpec IFU Mode

For the NIRSpec IFU mode, the overall processing flow is the same as other modes, in that the 1-D master background spectrum is created and applied during *calwebb_spec3* processing, as outlined above. No wavelength offset or bar shadow corrections are applied to IFU data, so any differences due to the way those calibrations are applied are not relevant to IFU mode. So the only effect that needs to be accounted for in the 2-D background generated from the master background is the difference between point source and uniform source pathloss corrections. This is accomplished by removing the uniform source pathloss correction from the 2-D background signal and applying the point source pathloss correction to it. It is then in a state where it matches the background signal contained in the point source IFU image from which it will be subtracted. Mathematically, the operation performed on the IFU 2-D background is:

$$bkg(corr) = bkg * pathloss(uniform)/pathloss(point)$$

The uniform and point source pathloss correction arrays referenced above are retrieved from the *cal* products used as input to the master background step. They are computed by the *pathloss* step during *calwebb_spec2* processing and stored as extra extensions in the *cal* products.

NIRSpec Fixed-Slit Mode

NIRSpec fixed slit data receive flat-field, pathloss, and photometric calibrations, all of which are wavelength-dependent, and the pathloss correction is also source type dependent. Fixed slit data do not receive a bar shadow correction. Only slits containing a point source can have a wavelength correction applied, to account for source centering within the slit, hence slits containing uniform sources receive the same flat-field and photometric calibrations as background spectra and therefore don't require corrections for those two calibrations. Furthermore, the source position in the slit is only known for the primary slit in an exposure, so even if the secondary slits contain point sources, no wavelength correction can be applied, and therefore again the flat-field and photometric calibrations are the same as for background spectra. This means only the pathloss correction difference between uniform and point sources needs to be accounted for in the secondary slits.

Therefore if the primary slit (as given by the FXD_SLIT keyword) contains a point source (as given by the SRCTYPE keyword) the corrections that need to be applied to the 2-D master background for that slit are:

$$\begin{aligned} bkg(corr) = bkg &* [flatfield(uniform)/flatfield(point)] \\ &* [pathloss(uniform)/pathloss(point)] \\ &* [photom(point)/photom(uniform)] \end{aligned}$$

For secondary slits that contain a point source, the correction applied to the 2-D master background is simply:

$$bkg(corr) = bkg * pathloss(uniform)/pathloss(point)$$

The uniform and point source versions of the flat-field, pathloss, and photom corrections are retrieved from the input *cal* product. They are computed and stored there during the execution of each of those steps during *calwebb_spec2* processing of NIRSpec Fixed-Slit exposures.

NIRSpec MOS Mode

Master background subtraction for NIRSpec MOS mode shares the high-level concepts of other modes, but differs greatly in the details. Most importantly, the source of the master background spectrum does not come from either nodded exposures or exposures of a background target. The background data instead come from designated background MSA slitlets contained with the same exposure as the science targets. Alternatively, a user can supply a master background spectrum to be used, as is the case for all other modes. The master background processing for MOS mode is therefore done within the *calwebb_spec2* pipeline when processing individual MOS exposures, rather than in the *calwebb_spec3* pipeline. Applying the master background subtraction within the *calwebb_spec2* pipeline also has advantages due to the complex series of operations that need to be performed, as described below.

During *calwebb_spec2* processing, all source and background slits are first partially calibrated up through the *extract_2d* and *srctype* steps of *calwebb_spec2*, which results in 2D cutouts for each slit with the source type identified. At this point the `master_background_mos` step is applied, which is a unique version of the step specifically tailored to NIRSpec MOS mode.

This version of the master background step completes the remaining calibration for all slits, but treats them all as extended sources and saves the correction arrays from each step (e.g. flat-field, pathloss, photom) for each slit, so that they can be used later to apply corrections to the background data. The resulting extracted 1D spectra from the background slits are combined to create the master background spectrum. The master background spectrum is then interpolated into the 2D space of each slit and has the photom, barshadow, pathloss, and flat-field corrections removed from the 2D background arrays, so that the background data now match the partially calibrated slit data from which they'll be subtracted. Mathematically, the corrections applied to the 2D master background for each MOS slit are:

$$bkg(corr) = bkg * flatfield(uniform) * pathloss(uniform) \\ * barshadow(uniform)/photom(uniform)$$

Once the corrected 2D backgrounds have been subtracted from each slit, processing returns to the *calwebb_spec2* flow, where all of the remaining calibration steps are applied to each slit, resulting in background-subtracted and fully calibrated 2D cutouts (*cal* and *s2d* products) and extracted 1D spectra (*x1d* products).

The detailed list of operations performed when applying master background subtraction to MOS data during *calwebb_spec2* processing is as follows:

1. Process all slitlets in the MOS exposure up through the *extract_2d* and *srctype* steps
2. The `master_background_mos` step temporarily applies remaining calibration steps up through *photom* to all slits, treating them all as extended sources (appropriate for background signal), and saving the extended source correction arrays for each slit in an internal copy of the data model
3. If a user-supplied master background spectrum is **not** given, the *resample_spec* and *extract_1d* steps are applied to the calibrated background slits, resulting in extracted 1D background spectra
4. The 1D background spectra are combined, using the *combine_1d* step, into a master background spectrum
5. If a user-supplied master background **is** given, steps 3 and 4 are skipped and the user-supplied spectrum is inserted into the processing flow
6. The master background spectrum (either user-supplied or created on-the-fly) is expanded into the 2D space of each slit
7. The 2D background "image" for each slit is processed in **inverse** mode through the *photom*, *barshadow*, *pathloss*, and *flatfield* steps, using the correction arrays that were computed in step 2, so that the background data now matches the partially calibrated background signal in each slit
8. The corrected 2D background is subtracted from each slit
9. The background-subtracted slits are processed through all remaining *calwebb_spec2* calibration steps, using the corrections appropriate for the source type in each slit

Step Arguments

The master background subtraction step uses the following optional arguments.

--user_background
The file name of a user-supplied 1-D master background spectrum. Must be in the form of a standard *x1d* product containing a single 'EXTRACT1D' extension. When a user background spectrum is supplied, it is used for the subtraction instead of a computed master background, and the name of the user-supplied file is recorded in the MSTRBKGD keyword in the output product(s). Defaults to `None`.

--save_background
A boolean indicating whether the computed 1-D master background spectrum should be saved to a file. The file name uses a product type suffix of "masterbg". If a user-supplied background is specified, this argument is ignored. Defaults to `False`.

--force_subtract
A boolean indicating whether or not to override the step's built-in logic for determining if the step should be applied. By default, the step will be skipped if the *calwebb_spec2 background* step has already been applied. If `--force_subtract = True`, the master background will be applied.

--output_use_model
A boolean indicating whether to use the "filename" meta attribute in the data model to determine the name of the output file created by the step. Defaults to `True`.

Reference Files

The master spectroscopic background subtraction step does not use any reference files.

jwst.master_background Package

Classes

MasterBackgroundStep([name, parent, ...])	MasterBackgroundStep: Compute and subtract master background from spectra
MasterBackgroundMosStep(*args, **kwargs)	Apply master background processing to NIRSpec MOS data

MasterBackgroundStep

class jwst.master_background.**MasterBackgroundStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

MasterBackgroundStep: Compute and subtract master background from spectra

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.

- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	Compute and subtract a master background spectrum

Attributes Documentation

class_alias = 'master_background'

spec

```
user_background = string(default=None) # Path to user-supplied master background
save_background = boolean(default=False) # Save computed master background
force_subtract = boolean(default=False) # Force subtracting master background
output_use_model = boolean(default=True)
```

Methods Documentation

process(*input*)

Compute and subtract a master background spectrum

Parameters

- **input** (ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.d IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.d ModelContainer, association) – Input target datamodel(s) to which master background subtraction is to be applied
- **user_background** (None, string, or MultiSpecModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSpecModel.html#jwst.datamodels.MultiS – Optional user-supplied master background 1D spectrum, path to file or opened datamodel
- **save_background** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Save computed master background.

- **force_subtract** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Optional user-supplied flag that overrides step logic to force subtraction of the master background. Default is False, in which case the step logic determines if the calspec2 background step has already been applied and, if so, the master background step is skipped. If set to True, the step logic is bypassed and the master background is subtracted.

Returns
result – The background-subtracted science datamodel(s)

Return type
ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels
IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.data
ModelContainer

MasterBackgroundMosStep

class jwst.master_background.**MasterBackgroundMosStep**(**args*, ***kwargs*)

Bases: JwstPipeline

Apply master background processing to NIRSpec MOS data

For MOS, and ignoring FS, the calibration process needs to occur twice: Once to calibrate background slits and create a master background. Then a second time to calibrate science using the master background.

Notes

The algorithm is as follows

- Calibrate all slits
  - For each step
    - Force the source type to be extended source for all slits.
    - Return the correction array used.
- Create the 1D master background
- For each slit
  - Expand out the 1D master background to match the 2D wavelength grid of the slit
  - Reverse-calibrate the 2D background, using the correction arrays calculated above.
  - Subtract the background from the input slit data

See Step.__init__ for the parameters.

Attributes Summary

class_alias
prefetch_references
spec
step_defs

Methods Summary

process(data)	Compute and subtract a master background spectrum
set_pars_from_parent()	Set substep parameters from the parents substeps

Attributes Documentation

class_alias = 'master_background_mos'

prefetch_references = False

spec

```
force_subtract = boolean(default=False)  # Force subtracting master background
save_background = boolean(default=False) # Save computed master background
user_background = string(default=None)   # Path to user-supplied master␣
↪background
inverse = boolean(default=False)    # Invert the operation
output_use_model = boolean(default=True)
```

step_defs = {'barshadow': <class 'jwst.barshadow.barshadow_step.BarShadowStep'>, 'flat_field': <class 'jwst.flatfield.flat_field_step.FlatFieldStep'>, 'pathloss': <class 'jwst.pathloss.pathloss_step.PathLossStep'>, 'photom': <class 'jwst.photom.photom_step.PhotomStep'>}

Methods Documentation

process(*data*)

Compute and subtract a master background spectrum

Parameters

data (MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jw – The data to operate on.

correction_pars

The master background information from a previous invocation of the step. Keys are:

- **"masterbkg_1d":** CombinedSpecModel **(https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CombinedSpecModel.html#jwst.datamode** The 1D version of the master background.

- **"masterbkg_2d": MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamodels.Mul**
  The 2D slit-based version of the master background.

  Type
  dict (https://docs.python.org/3/library/stdtypes.html#dict)

force_subtract

Optional user-supplied flag that overrides step logic to force subtraction of the master background. Default is False, in which case the step logic determines if the calspec2 background step has already been applied and, if so, the master background step is skipped. If set to True, the step logic is bypassed and the master background is subtracted.

Type
bool (https://docs.python.org/3/library/functions.html#bool), optional

save_background

Save computed master background.

Type
bool (https://docs.python.org/3/library/functions.html#bool), optional

user_background

Optional user-supplied master background 1D spectrum, path to file or opened datamodel

Type
None, string, or CombinedSpecModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.Combin

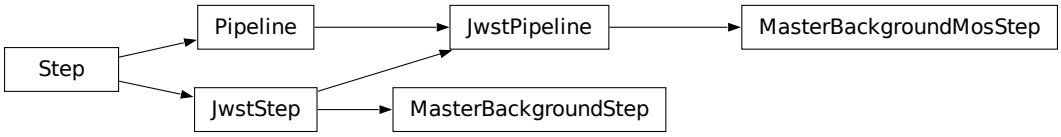
Returns
result

Return type
MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.dat

set_pars_from_parent()

Set substep parameters from the parents substeps

Class Inheritance Diagram



15.1.36 Model Blender

Role of Model Blender

One problem with combining data from multiple exposures stems from not being able to keep track of what kind of data was used to create the final product. The final product only reports one value for each of the metadata attributes from the model schema used to describe the science data, where each of multiple inputs may have had different values for each attribute. The model_blender package solves this problem by allowing the user to define rules that can be used to determine a final single value from all the input values for each attribute, using separate rules for each attribute as appropriate. This package also creates a FITS binary table that records the input attribute values for all the input models used to create the final product, allowing the user to select what attributes to keep in this table.

This code works by

- reading in all the input datamodels (either already in-memory or from FITS files)
- evaluating the rules for each attribute as defined in the model's schema
- determining from definitions in the input model's schema what attributes to keep in the table
- applying each attributes rule to the set of input values to determine the final output value
- updating the output model's metadata with the new values
- generating the output table with one row for each input model's values

Using model_blender

The model blender package requires

- all the input models be available
- the output product has already been generated

Both the input models and output product could be provided as either a datamodel instance in memory or as the name of a FITS file on disk. The primary advantage to working strictly in-memory with datamodel instances comes from minimizing the amount of disk I/O needed for this operation which can result in significantly more efficient (read that: faster) processing.

Note: The generated output model will be considered to contain a default (or perhaps even empty) set of Metadata (https://stdatamodels.readthedocs.io/en/latest/jwst/datamodels/metadata.html#metadata) based on some model defined in stdatamodels (https://stdatamodels.readthedocs.io/en/latest/jwst/datamodels/index.html#data-models) This metadata will be replaced **in-place** when running *Model Blender*.

The simplest way to run model blender only requires calling a single interface:

```
from jwst.model_blender import blendmeta
blendmeta.blendmodels(product, inputs=input_list)
```

where

- `product`: the datamodel (or FITS filename) for the already combined product
- `input_list`: list of input datamodels or FITS filenames for all inputs used to create the `product`

The output product will end up with new metadata attribute values and a new HDRTAB FITS binary table extension in the FITS file when the product gets saved to disk.

Customizing the behavior

By default, `blendmodels` will not write out the updated product model to disk. This allows the user or calling program to revise or apply data-specific logic to redefine the output value for any of the output product's metadata attributes. For example, when combining multiple images, the WCS information does not represent any combination of the input WCS attributes. Instead, the user can have their own processing code replace the *blended* WCS attributes with one that was computed separately using a complex, accurate algorithm. This is, in fact, what the resample step does to create the final resampled output product whenever it is called by steps in the JWST pipeline.

Additional control over the behavior of model_blender comes from editing the schema for the input datamodels where the rules for each attribute are defined. A sample definition from the core schema demonstrates the basic syntax used for any model blending definitions:

```
time_end:
  title: UTC time at end of exposure
  type: string
  fits_keyword: TIME-END
  blend_rule: last
  blend_table: True
```

Any attribute without an entry for `blend_rule` will use the default rule of `first` which selects the first value from all inputs in the order provided as the final output value. Any attribute with a `blend_table` rule will insure that the specific attribute will be included in the output HDRTAB binary table appended to the product model when it gets written out to disk as a FITS file.

The full set of rules included in the package are described in *Model Blender Rules* and include common list/array operations such as (but not limited to):

- minimum
- maximum
- first
- last
- mean
- zero

These can then be used to customize the output value for any given attribute should the rule provided by default with the schema installed with the JWST environment not be correct for the user's input data. The user can simply edit the schema definition installed in their JWST environment to apply custom rules for blending the data being processed.

Model Blender

These functions serve as the primary interface for blending models.

jwst.model_blender.blendmeta Module

blendmeta - Merge metadata from multiple models.

This module will create a new metadata instance and table from a list of input datamodels or filenames.

Functions

blendmodels(product[, inputs, output, ...])	Run main interface for blending metadata from multiple models.
build_tab_schema(new_table)	Return new schema definition that describes the input table.
cat_headers(hdr1, hdr2)	Create new `astropy.io.fits.Header` object from concatenating 2 input Headers
convert_dtype(value)	Convert numarray column dtype into YAML-compatible format description
extract_filenames_from_product(product)	Returns the list of filenames with extensions of input observations that were used to generate the product.
get_blended_metadata(input_models[, verbose])	Return a blended metadata instance and table based on the input datamodels.

blendmodels

jwst.model_blender.blendmeta.**blendmodels**(*product*, *inputs=None*, *output=None*, *ignore=None*, *verbose=False*)

Run main interface for blending metadata from multiple models.

Blend models that went into creating the original drzfile into a new metadata instance with a table that contains attribute values from all input datamodels.

The product will be used to determine the names of the input models, should no filenames be provided in the 'inputs' parameter.

The product will be updated 'in-place' with the new metadata attributes and FITS BinTableHDU table. The blended FITS table, with extname=HDRTAB, has 1 column for each metadata attribute recorded from the input models, one row for each input model, and column names are the FITS keywords for that metadata attribute. For example, values from `meta.observation.time` would be stored in the `TIME-OBS` column.

Rules for what function to use to determine the blended output attribute value and what metadata attributes should be used as columns in the blended FITS table are defined in the datamodel schema.

Note: Custom rules for a metadata value should be computed by the calling routine and used to update the metadata in the output model AFTER calling this function.

Parameters

- **product** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Name of combined product with metadata that needs updating. This can be specified as a single filename. When no value for `inputs` has been provided, this file will also evaluate `meta.asn` to determine the names of the input datamodels whose metadata need to be blended to create the new combined metadata.

- **inputs** (*list* (https://docs.python.org/3/library/stdtypes.html#list), *optional*) – This can be either a list of filenames or a list of DataModels objects. If provided, the filenames provided in this list will be used to get the metadata which will be blended into the final output metadata.
- **output** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – If provided, update `meta.filename` in the blended `product` to define what file this model will get written out to.
- **ignore** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of str* (https://docs.python.org/3/library/stdtypes.html#str), *None*, *optional*) – A list of string the meta attribute names which, if provided, will show which attributes should not be blended.
- **verbose** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional [Default: False]*) – Print out additional messages during processing when specified.

Example

This example shows how to blend the metadata from a set of DataModels already read in memory for the product created by the `resample` step. This example relies on the Association file used as the input to the `resample` step to specify all the inputs for blending using the following syntax:

```
>>> from stdatamodels.jwst import datamodels
>>> asnfile = "jw99999-a3001_20170327t121212_coron3_001_asn.json"
>>> data = datamodels.open(asnfile)
>>> input_models = [data[3], data[4]]  # we know the last datasets are SCIENCE
>>> blendmodels(data.meta.asn_table.products[0].name, inputs=input_models)
```

build_tab_schema

jwst.model_blender.blendmeta.**build_tab_schema**(*new_table*)

Return new schema definition that describes the input table.

cat_headers

jwst.model_blender.blendmeta.**cat_headers**(*hdr1*, *hdr2*)

Create new `astropy.io.fits.Header` object from concatenating 2 input Headers

convert_dtype

jwst.model_blender.blendmeta.**convert_dtype**(*value*)

Convert numarray column dtype into YAML-compatible format description

extract_filenames_from_product

jwst.model_blender.blendmeta.**extract_filenames_from_product**(*product*)

Returns the list of filenames with extensions of input observations that were used to generate the product.

get_blended_metadata

jwst.model_blender.blendmeta.**get_blended_metadata**(*input_models*, *verbose=False*)

Return a blended metadata instance and table based on the input datamodels. This will serve as the primary interface for blending datamodels.

Parameters

input_models (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – Either a single list of filenames from which to extract the metadata to be blended, or a list of `datamodels.JwstDataModel` objects to be blended. The input models are assumed to have the blending rules defined as an integral part of the schema definition for the model.

Returns

- **metadata** (*list*) – A list of blended metadata instances, one for each i
- **new_table** (*object*) – Single fits.TableHDU object that contains the combined results from all input headers(extension). Each row will correspond to an image, and each column corresponds to a single keyword listed in the rules.

jwst.model_blender.blender Module

Functions

metablender(input_models, spec)	Given a list of datamodels, aggregate metadata attribute values and create a table made up of values from a number of metadata instances, according to the given specification.

metablender

jwst.model_blender.blender.**metablender**(*input_models*, *spec*)

Given a list of datamodels, aggregate metadata attribute values and create a table made up of values from a number of metadata instances, according to the given specification.

Parameters:

- *input_models* is a sequence where each element is either:
  - a `datamodels.JwstDataModel` instance or sub-class
  - a string giving the *filename* for the input_model
- *spec* is a list defining which keyword arguments are to be aggregated and how. Each element in the list should be a sequence with 2 to 5 elements of the form:

  (*src_keyword*, *dst_name*, *function*, *error_type*, *error_value*)

- *src_keyword* is the keyword to pull values from. It is case-insensitive.
- *dst_name* is the name to use as a dictionary key or column name for the destination values.
- *function* (optional). If function is not None, the values from the source are aggregated and returned in the *aggregate_dict*. If function is None (or the tuple contains only 2 elements), all values are stored as a column with the name *dst_name* in the result *table*.

  If not None, *function* should be a callable object that takes a sequence of values and returns an aggregate result. If the function returns None, no values will be added to the aggregate dictionary. There are many functions in Numpy that are directly useful as an aggregating function, for example:

  * mean: `numpy.mean` (https://numpy.org/devdocs/reference/generated/numpy.mean.html#numpy.mean)
  * median: `numpy.median` (https://numpy.org/devdocs/reference/generated/numpy.median.html#numpy.median)
  * maximum: `numpy.max` (https://numpy.org/devdocs/reference/generated/numpy.max.html#numpy.max)
  * minimum: `numpy.min` (https://numpy.org/devdocs/reference/generated/numpy.min.html#numpy.min)
  * sum: `numpy.sum` (https://numpy.org/devdocs/reference/generated/numpy.sum.html#numpy.sum)
  * standard deviation: `numpy.std` (https://numpy.org/devdocs/reference/generated/numpy.std.html#numpy.std)

  Lambda functions are also often useful:

  * first: `lambda x: x[0]`
  * last: `lambda x: x[-1]`

  Additionally, *function* may be a tuple, where each member is itself a callable object. The result will be a tuple containing results from each of the given functions. For instance, to aggregate a range of values, i.e. both the minimum and maximum values, use the following as *function*: `(numpy.min, numpy.max)`.
- *error_type* (optional) defines how missing or syntax-errored values are handled. It may be one of the following:

  * 'ignore': missing or unparsable values are ignored. They are not included in the list of values passed to the aggregating function. In the result *table*, missing values are masked out.
  * 'raise': missing or unparsable values raise a `ValueError` (https://docs.python.org/3/library/exceptions.html#ValueError) exception.
  * 'constant': missing or unparsable values are replaced with a constant, given by the *error_value* field.
- *error_value* (optional) is the constant value to be used for missing or unparsable values when *error_type* is set to 'constant'. When not provided, it defaults to `NaN`.

Returns:

A 2-tuple of the form (*aggregate_dict*, *table*) where:

- *aggregate_dict* is a dictionary of where the keys come from *dst_name* and the values are the aggregated values as run_KeywordMapping through *function*.
- *table* is a masked Numpy structured array where the column names come from *dst_name* and the column contains the values from *src_keyword* for all of the given headers. Missing values are masked out.

Model Blender Rules

Blending models relies on rules to define how to evaluate all the input values for a model attribute in order to determine the final output value. These rules then get specified in the model schema for each attribute.

The rules get interpreted and applied as list or array operations that work on the set of input values for each attribute. The full set of pre-defined rules includes

```
import numpy as np
# translation dictionary for function entries from rules files
blender_funcs = {'first': first,
                 'last': last,
                 'float_one': float_one,
                 'int_one': int_one,
                 'zero': zero,
                 'multi': multi,
                 'multi?': multi1,
                 'mean': np.mean,
                 'sum': np.sum,
                 'max': np.max,
                 'min': np.min,
                 'stddev': np.std,
                 'mintime': mintime,
                 'maxtime': maxtime,
                 'mindate': mindate,
                 'maxdate': maxdate,
                 'mindatetime': mindatetime,
                 'maxdatetime': maxdatetime}
```

The rules that should be referenced in the model schema definition are the keys defined for `jwst.model_blender.blender_rules.blender_funcs` listed above. This definition illustrates how several rules are simply interfaces for numpy array operations, while others are defined internally to `model_blender`.

jwst.model_blender.blendrules Module

blendmeta - Merge metadata from multiple models to create a new metadata instance and table

Functions

find_keywords_in_section(hdr, title)	Return a list of keyword names.
first(items)	Return first item from list of values
float_one(vals)	Return a constant floating point value of 1.0
int_one(vals)	Return an integer value of 1
interpret_attr_line(attr, line_spec)	Generate rule for single attribute from input line from rules file.
interpret_entry(line, hdr)	Generate the rule(s) specified by the entry from the rules file.
last(items)	Return last item from list of values
maxdate(items)	Return the maximum date from a list of date strings in yyyy-mm-dd format.
maxdatetime(items)	Return the maximum datetime from a list of datetime strings in ISO-8601 format.
maxtime(items)	
mindate(items)	Return the minimum date from a list of date strings in yyyy-mm-dd format.
mindatetime(items)	Return the minimum datetime from a list of datetime strings in ISO-8601 format.
mintime(items)	
multi(vals)	This will either return the common value from a list of identical values or 'MULTIPLE'
multi1(vals)	This will either return the common value from a list of identical values or the single character '?'
zero(vals)	Return a value of 0

find_keywords_in_section

jwst.model_blender.blendrules.**find_keywords_in_section**(*hdr*, *title*)

Return a list of keyword names.

The list will be derived from the section
with the specified section title identified in the hdr.

first

jwst.model_blender.blendrules.**first**(*items*)

Return first item from list of values

float_one

jwst.model_blender.blendrules.**float_one**(*vals*)

Return a constant floating point value of 1.0

int_one

jwst.model_blender.blendrules.**int_one**(*vals*)

Return an integer value of 1

interpret_attr_line

jwst.model_blender.blendrules.**interpret_attr_line**(*attr*, *line_spec*)

Generate rule for single attribute from input line from rules file.

interpret_entry

jwst.model_blender.blendrules.**interpret_entry**(*line*, *hdr*)

Generate the rule(s) specified by the entry from the rules file.

Notes

The entry should always be a dict with format: {attribute_name : {'rule':'some_rule', 'output':''}} – or (for table column specification)– {attribute_name: attribute_name} where 'output' is assumed to be the same as attribute_name if not present

last

jwst.model_blender.blendrules.**last**(*items*)

Return last item from list of values

maxdate

jwst.model_blender.blendrules.**maxdate**(*items*)

Return the maximum date from a list of date strings in yyyy-mm-dd format.

maxdatetime

jwst.model_blender.blendrules.**maxdatetime**(*items*)

Return the maximum datetime from a list of datetime strings in ISO-8601 format.

maxtime

jwst.model_blender.blendrules.**maxtime**(*items*)

mindate

jwst.model_blender.blendrules.**mindate**(*items*)

Return the minimum date from a list of date strings in yyyy-mm-dd format.

mindatetime

jwst.model_blender.blendrules.**mindatetime**(*items*)

Return the minimum datetime from a list of datetime strings in ISO-8601 format.

mintime

jwst.model_blender.blendrules.**mintime**(*items*)

multi

jwst.model_blender.blendrules.**multi**(*vals*)

This will either return the common value from a list of identical values or 'MULTIPLE'

multi1

jwst.model_blender.blendrules.**multi1**(*vals*)

This will either return the common value from a list of identical values or the single character '?'

zero

jwst.model_blender.blendrules.**zero**(*vals*)

Return a value of 0

Classes

KeywordRules(model)	Read in the rules used to interpret the keywords from the specified instrument image header.
KwRule(line)	This class encapsulates the logic needed for interpreting a single keyword rule from a text file.

KeywordRules

class jwst.model_blender.blendrules.**KeywordRules**(*model*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Read in the rules used to interpret the keywords from the specified instrument image header.

Methods Summary

add_rules_kws(hdr)	Update metadata with .. warning::.
apply(models[, tabhdu])	For a full list of metadata objects, apply the specified rules to generate a dictionary of new values and a table using blender.
index_of(kw)	Reports the index of the specified kw.
interpret_rules(hdrs)	Convert specifications for rules from rules file into specific rules for this header(instrument/detector).
merge(kwrules)	Merge a new set of interpreted rules into the current set The new rules, kwrules, can either be a new class or a whole new set of rules (like those obtained from using self.interpret_rules with a new header).

Methods Documentation

add_rules_kws(*hdr*)

Update metadata with .. warning:

```
Needs to be modified to work with metadata.

Update PRIMARY header with HISTORY cards that report the exact
rules used to create this header. Only non-comment lines from the
rules file will be reported.
```

apply(*models*, *tabhdu=False*)

For a full list of metadata objects, apply the specified rules to generate a dictionary of new values and a table using blender.

This method returns the new metadata object and summary table as `datamodels.model.ndmodel` and fits.binTableHDU objects.

index_of(*kw*)

Reports the index of the specified kw.

interpret_rules(*hdrs*)

Convert specifications for rules from rules file into specific rules for this header(instrument/detector).

Notes

This allows for expansion rules to be applied to rules from the rules files (such as any wildcards or section titles).

Output will be 'self.rules' that contains a list of tuples: - a tuple of 2 values for each column in the table - a tuple of 4 values for each attribute identified in metadata Partial sample from HST to show format: [('CTYPE1O', 'CTYPE1O'), ('CTYPE2O', 'CTYPE2O'), ('CUNIT1O', 'CUNIT1O'), ('CUNIT2O', 'CUNIT2O'), ('APERTURE', 'APERTURE', <function fitsblender.blendheaders.multi>, 'ignore'), ('DETECTOR', 'DETECTOR', <function fitsblender.blender.first>, 'ignore'), ('EXPEND', 'EXPEND', <function numpy.core.fromnumeric.amax>, 'ignore'), ('EXPSTART', 'EXPSTART', <function numpy.core.fromnumeric.amin>, 'ignore'), ('EXPTIME', 'TEXPTIME', <function numpy.core.fromnumeric.sum>, 'ignore'), ('EXPTIME', 'EXPTIME', <function numpy.core.fromnumeric.sum>, 'ignore')]

This rules format will allow the algorithm, logic and code from the original fitsblender to be used with as little change as possible. It will need to be derived (as with HST) from the input models metadata for expansion of attribute sections or wildcards in attributes specified in the rules.

merge(*kwrules*)

Merge a new set of interpreted rules into the current set The new rules, kwrules, can either be a new class or a whole new set of rules (like those obtained from using self.interpret_rules with a new header).

KwRule

class jwst.model_blender.blendrules.**KwRule**(*line*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

This class encapsulates the logic needed for interpreting a single keyword rule from a text file.

Notes

The `.rules` attribute contains the interpreted set of rules that corresponds to this line.

Example:

```
Interpreting rule from
{'meta.attribute': { 'rule': 'first', 'output': 'meta.attribute'}}
--or--
{'meta.attribute': 'meta.attribute'}  # Table column specification

into rule [('meta.attribute', 'meta.attribute', <function first at 0x7fe505db7668>,
↪'ignore')]
and sname None
```

Initialize new keyword rule.

Parameters

line (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Line should be dict with attribute name as the key, and a dict as the value specifying 'rule' and (optionally)'output'.

Methods Summary

interpret(hdr)	Use metadata to interpret rule.

Methods Documentation

interpret(*hdr*)

Use metadata to interpret rule.

jwst.model_blender Package

15.1.37 MIRI MRS Sky Matching

Description

Class

jwst.mrs_imatch.MRSIMatchStep

Alias

mrs_imatch

Overview

The `mrs_imatch` step "matches" image intensities of several input 2D MIRI MRS images by fitting polynomials to cube intensities (cubes built from the input 2D images), in such a way as to minimize - in the least squares sense - inter-image mismatches in intensity. The "background matching" polynomials are defined in the frame of world coordinates (e.g. RA, DEC, lambda).

If any of background polynomial coefficients are a nan then the step is skipped and S_MRSMAT is set to SKIPPED.

Any sources in the scene are identified via sigma clipping and removed from the matching region.

Assumptions

Because the fitted polynomials are defined in terms of world coordinates, and because the algorithm needs to build 3D cubes for each input image, all input images need to have a valid WCS defined.

Algorithm

This step builds a system of linear equations

$$a \cdot c = b$$

whose solution c is a set of coefficients of (multivariate) polynomials that represent the "background" in each input image (these are polynomials that are "corrections" to the intensities in the input images) such that the following sum is minimized:

$$L = \sum_{n,m=1,n\neq m}^{N} \sum_{k} \frac{[I_n(k) - I_m(k) - P_n(k) + P_m(k)]^2}{\sigma_n^2(k) + \sigma_m^2(k)}.$$

In the above equation, index $k = (k_1, k_2, ...)$ labels a position in an input image's pixel grid [NOTE: all input images share a common pixel grid].

"Background" polynomials $P_n(k)$ are defined through the corresponding coefficients as:

$$P_n(k_1, k_2, ...) = \sum_{d_1=0, d_2=0, ...}^{D_1, D_2, ...} c^n_{d_1, d_2, ...} \cdot k_1^{d_1} \cdot k_2^{d_2} \cdot ....$$

Step Arguments

The `mrs_imatch` step has two optional arguments:

`bkg_degree`
: The background polynomial degree (int; default=1)

`subtract`
: Indicates whether the computed matching "backgrounds" should be subtracted from the image data (bool; default=False)

Reference Files

This step does not require any reference files.

Also See

See wiimatch package documentation (http://wiimatch.readthedocs.io) for more details.

Also See:

LSQ Equation Construction and Solving

jwst.mrs_imatch.mrs_imatch_step Module

JWST pipeline step for image intensity matching for MIRI images.

Authors
: Mihai Cara

Functions

apply_background_2d(model2d[, channel, subtract])	Apply (subtract or add back) background values computed from `meta.background` polynomials to 2D image data.

apply_background_2d

jwst.mrs_imatch.mrs_imatch_step.**apply_background_2d**(*model2d*, *channel=None*, *subtract=True*)

Apply (subtract or add back) background values computed from `meta.background` polynomials to 2D image data.

This function modifies the input `model2d`'s data.

Warning: This function does not check whether background was previously applied to image data (through `meta.background.subtracted`).

Warning: This function does not modify input model's `meta.background.subtracted` attribute to indicate that background has been applied to model's data. User is responsible for setting `meta.background.subtracted` after background was applied to all channels. Partial application of background (i.e., to only *some channels* as opposite to *all* channels) is not recommended.

Parameters

- **model2d** (jwst.datamodels.image.ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamod – A jwst.datamodels.image.ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.Im from whose data background needs to be subtracted (or added back).
- **channel** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *int* (https://docs.python.org/3/library/functions.html#int), *list* (https://docs.python.org/3/library/stdtypes.html#list), *None*, *optional*) – This parameter indicates for which channel background values should be applied. An integer value is automatically converted to a string type. A string type input value indicates **a single** channel to which background should be applied. `channel` can also be a list of several string or integer single channel values. The default value of None (https://docs.python.org/3/library/constants.html#None) indicates that background should be applied to all channels.
- **subtract** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Indicates whether to subtract or add back background values to input model data. By default background is subtracted from data.

Classes

MRSIMatchStep([name, parent, config_file, ...])	MRSIMatchStep: Subtraction or equalization of sky background in MIRI MRS science images.

MRSIMatchStep

class jwst.mrs_imatch.mrs_imatch_step.**MRSIMatchStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

MRSIMatchStep: Subtraction or equalization of sky background in MIRI MRS science images.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(images)	This is where real work happens.

Attributes Documentation

class_alias = 'mrs_imatch'

reference_file_types = []

spec

```
# General sky matching parameters:
bkg_degree = integer(min=0, default=1) # Degree of the polynomial for␣
↪background fitting
subtract = boolean(default=False) # subtract computed sky from 'images' cube␣
↪data?
```

Methods Documentation

process(*images*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

jwst.mrs_imatch Package

This package provides support for image intensity subtraction and equalization (matching) for MIRI images.

Classes

MRSIMatchStep([name, parent, config_file, ...])	MRSIMatchStep: Subtraction or equalization of sky background in MIRI MRS science images.

MRSIMatchStep

class jwst.mrs_imatch.**MRSIMatchStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

MRSIMatchStep: Subtraction or equalization of sky background in MIRI MRS science images.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(images)	This is where real work happens.

Attributes Documentation

class_alias = 'mrs_imatch'

reference_file_types = []

spec

```
# General sky matching parameters:
bkg_degree = integer(min=0, default=1) # Degree of the polynomial for␣
↪background fitting
subtract = boolean(default=False) # subtract computed sky from 'images' cube␣
↪data?
```

Methods Documentation

process(*images*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.38 MSAFlagOpen Correction

Description

Class

jwst.msaflagopen.MSAFlagOpenStep

Alias

msa_flagging

Overview

The `msaflagopen` step flags pixels in NIRSpec exposures that are affected by MSA shutters that are stuck in the open position.

Background

The correction is applicable to NIRSpec IFU and MSA exposure types.

Algorithm

The set of shutters whose state is not commandable (i.e. they are permanently stuck in 'open' or 'closed' positions) is recorded in the MSAOPER reference file. The reference file is searched for all shutters with any of the quantities 'Internal state', 'TA state' or 'state' set to 'open'.

The step loops over the list of stuck open shutters. For each shutter, the bounding box that encloses the projection of the shutter onto the detector array is calculated, and for each pixel in the bounding box, the WCS is calculated. If the pixel is inside the region affected by light through the shutter, the WCS will have valid values, whereas if the pixel is outside, the WCS values will be NaN. The indices of each non-NaN pixel in the WCS are used to alter the corresponding pixels in the DQ array by OR'ing their DQ value with that for "MSA_FAILED_OPEN."

Step Arguments

The `msaflagopen` step has no step-specific arguments.

Reference File

The `msaflagopen` step uses a MSAOPER reference file.

MSAOPER Reference File

REFTYPE
MSAOPER

Data model
N/A

The MSAOPER reference file contains a list of failed MSA shutters and their failure state (stuck open, stuck closed, etc.)

Reference Selection Keywords for MSAOPER

CRDS selects appropriate MSAOPER references based on the following keywords. MSAOPER is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for MSAOPER

In addition to the standard reference file keywords listed above, the following keywords are *required* in MSAOPER reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for MSAOPER*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

Reference File Format

The MSAOPER reference files are json format.

The fields are:

title
Short description of the reference file

reftype
Should be "MSAOPER"

pedigree
Should be one of "DUMMY", "GROUND" or "INFLIGHT"

author
Creator of the file

instrument
JWST Instrument; should be "NIRSPEC"

exp_type
EXP_TYPEs this file should be used with; should be "NRS_IFU|NRS_MSASPEC"

telescope
Should be "JWST"

useafter
Exposure datetime after which this file is applicable

descrip
Description of reference file

msaoper

Q
Quadrant; should be an integer 1-4

x
x location of shutter (integer, 1-indexed)

y
y location of shutter (integer, 1-indexed)

state
state of shutter; should be "closed" or "open"

TA state
TA state of shutter; should be "closed" or "open"

Internal state
Internal state of shutter; should be "closed", "normal" or "open"

Vignetted
Is the shutter vignetted? Should be "yes" or "no"

history
Description of the history relevant to this file; might point to documentation

jwst.msaflagopen Package

Classes

MSAFlagOpenStep([name, parent, config_file, ...])	MSAFlagOpenStep: Flags pixels affected by MSA failed open shutters

MSAFlagOpenStep

class jwst.msaflagopen.**MSAFlagOpenStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

MSAFlagOpenStep: Flags pixels affected by MSA failed open shutters

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'msa_flagging'

reference_file_types = ['msaoper']

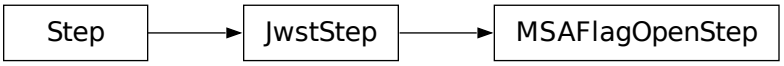
spec

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.39 NSClean 1/f Correction

Description

Class

jwst.nsclean.NSCleanStep

Alias

nsclean

Overview

The `nsclean` step applies an algorithm for removing correlated read noise from NIRSpec images. The noise often appears as faint vertical banding and so-called "picture frame noise." The algorithm uses dark (unilluminated) areas of an image to fit a background model in Fourier space. When the fit is subtracted, it removes nearly all correlated noise. Compared to simpler strategies, like subtracting a rolling median, this algorithm is more thorough and uniform. It is also computationally undemanding, typically requiring only a few seconds to clean a full-frame image.

The correction can be applied to any type of NIRSpec exposure, including IFU, MOS, fixed slit, and Bright Object Time Series (BOTS), in both full-frame and subarray readouts. Time series (3D) data are corrected one integration at a time.

Note: The step is currently not capable of processing images taken using the "ALLSLITS" subarray. Other subarray types are allowed.

Details on the source of the correlated noise and the algorithm used in the `nsclean` step to fit and remove it can be found in Rauscher 2023 (https://ui.adsabs.harvard.edu/abs/2023arXiv230603250R/abstract).

Upon completion of the step, the step status keyword "S_NSCLEN" gets set to "COMPLETE" in the output science data.

Assumptions

As described below, the creation of a pixel mask depends on the presence of a World Coordinate System (WCS) object for the image, which is constructed by the *assign_wcs* step. In addition, creating a mask for IFU and MOS images depends on the presence of DQ flags assigned by the *msaflagopen* step. It is therefore required that those steps be run before attempting to apply `nsclean`.

Creation of an image mask

One of the key components of the correction is knowing which pixels are unilluminated and hence can be used in fitting the background noise. The step builds a mask on the fly for each image, which is used to mark useable and unuseable pixels. The mask is a 2D boolean array, having the same size as the image, with pixels set to True interpreted as being OK to use. The process of building the mask varies somewhat depending on the observing mode of the image being processed. Some features are common to all modes, while others are mode-specific. The following sections describe each type of masking that can be applied and at the end there is a summary of the types applied to each image mode.

The user-settable step parameter `save_mask` can be used to save the mask to a file, if desired (see *nsclean step arguments*).

Note that a user may supply their own mask image as input to the step, in which case the process of creating a mask is skipped. The step parameter `user_mask` is used to specify an input mask.

IFU Slices

For IFU images the majority of the mask is based on knowing which pixels are contained within the footprints of the IFU slices. To do this, the image's World Coordinate System (WCS) object is queried in order to determine which pixels are contained within each of the 30 slices. Pixels within each slice are set to False (do not use) in the mask.

MOS/FS Slits

The footprints of each open MOS slitlet or fixed slit are flagged in a similar way as IFU slices. For MOS and FS images, the WCS object is queried to determine which pixels are contained within each open slit/slitlet and they are set to False in the mask.

MSA Failed Open Shutters

Pixels affected by stuck open MSA shutters are masked, because they may contain signal. This is accomplished by setting all pixels flagged by the *msaflagopen* step with DQ value "MSA_FAILED_OPEN" to False in the mask.

NaN Pixels

Any pixel in the input image that has a value of NaN is temporarily reset to zero for input to the fitting routine and flagged as False in the mask. Upon completion of the noise subtraction, this population of pixels is set back to NaN again in the output (corrected) image.

Fixed-Slit Region Pixels

Full-frame MOS and IFU images may contain signal from the always open fixed slits, which appear in fixed region in the middle of each image. The entire region containing the fixed slits is masked out when processing MOS and IFU images. The masked region is currently hardwired in the step to image indexes [1:2048, 923:1116], where the indexes are in x, y order and in 1-indexed values.

Left/Right Reference Pixel Columns

Full-frame images contain 4 columns of reference pixels on the left and right edges of the image. These are not to be used in the fitting algorithm and hence are set to False in the mask.

Outliers

Pixels in the unilluminated regions of the region can contain anomalous signal, due to uncaught Cosmic Rays, hot pixels, etc. A sigma-clipping routine is employed to find such outliers within the input image and set them to False in the mask. All pixels with values greater than $median + n_sigma * sigma$ are set to False in the mask. Here `median` and `sigma` are computed from the image using the astropy.stats `sigma_clipped_stats` routine, using the image mask to exclude pixels that have already been flagged and a clipping level of 5 sigma. `n_sigma` is a user-settable step parameter, with a default value of 5.0 (see *nsclean step arguments*).

Mode-Specific Masking Steps

The following table indicates which flavors of masking are applied to images from each type of observing mode.

Masking	Mode IFU	MOS	FS
IFU Slices[1]	✓		
Slits/Slitlets[1]		✓	✓
MSA_FAILED_OPEN	✓	✓	✓
NaN Pixels	✓	✓	✓
FS Region	✓	✓	
Reference Pix	✓	✓	✓
Outliers	✓	✓	✓

[1]The application of these steps can be turned on and off via the step parameter `mask_spectral_regions`. This parameter controls whether the "IFU Slices" and "Slits/Slitlets" portions of the masking are applied.

Reference Files

The `nsclean` step does not use any reference files.

Step Arguments

The `nsclean` step has the following optional arguments to control the behavior of the processing.

`--mask_spectral_regions` (boolean, default=True)
Mask regions in IFU and MOS images that are within the bounding boxes for each slice or slitlet defined in the WCS object of the image.

`--n_sigma` (float, default=5.0)
The sigma-clipping threshold to use when searching for outliers and illuminated pixels to be excluded from use in the fitting process.

`--save_mask` (boolean, default=False)
A flag to indicate whether the mask constructed by the step should be saved to a file.

`--user_mask` (string, default=None)
Path to a user-supplied mask file. If supplied, the mask is used directly and the process of creating a mask in the step is skipped.

jwst.nsclean Package

Classes

NSCleanStep([name, parent, config_file, ...])	NSCleanStep: This step performs 1/f noise correction ("cleaning") of NIRSpec images, using the "NSClean" method.

NSCleanStep

class jwst.nsclean.**NSCleanStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

NSCleanStep: This step performs 1/f noise correction ("cleaning") of NIRSpec images, using the "NSClean" method.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	Fit and subtract 1/f background noise from a NIRSpec image

Attributes Documentation

class_alias = 'nsclean'

spec

```
mask_spectral_regions = boolean(default=True)  # Mask WCS-defined regions
n_sigma = float(default=5.0)  # Clipping level for outliers
save_mask = boolean(default=False)  # Save the created mask
user_mask = string(default=None)  # Path to user-supplied mask
skip = boolean(default=True)  # By default, skip the step
```

Methods Documentation

process(*input*)

Fit and subtract 1/f background noise from a NIRSpec image

Parameters

- **input** (ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.d IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.d – Input datamodel to be corrected
- **n_sigma** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Sigma clipping threshold to be used in detecting outliers in the image
- **save_mask** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Save the computed mask image
- **user_mask** (None, str, or ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageM – Optional user-supplied mask image; path to file or opened datamodel
- **mask_spectral_regions** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Mask regions of the image defined by WCS bounding boxes for slits/slices

Returns

output_model – The 1/f corrected datamodel

Return type

ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.data

Class Inheritance Diagram

15.1.40 Outlier Detection

Description

Classes

jwst.outlier_detection.OutlierDetectionStep, *jwst.outlier_detection.OutlierDetectionScaledStep*, *jwst.outlier_detection.OutlierDetectionStackStep*

Aliases

outlier_detection, outlier_detection_scaled, outlier_detection_stack

Processing multiple datasets together allows for the identification of bad pixels or cosmic-rays that remain in each of the input images, many times at levels which were not detectable by the *jump* step. The `outlier_detection` step implements the following algorithm to identify and flag any remaining cosmic-rays or other artifacts left over from previous calibrations:

1. build a stack of input data
    - all inputs will need to have the same WCS since outlier detection assumes the same flux for each point on the sky, and variations from one image to the next would indicate a problem with the detector during readout of that pixel
    - if needed, each input will be resampled to a common output WCS
2. create a median image from the stack of input data
    - this median operation will ignore any input pixels which have a weight which is too low (<70% max weight)
3. create "blotted" data from the median image to exactly match each original input dataset
4. perform a statistical comparison (pixel-by-pixel) between the median blotted data with the original input data to look for pixels with values that are different from the mean value by more than some specified sigma based on the noise model
    - the noise model used relies on the error array computed by previous calibration steps based on the readnoise and calibration errors
5. flag the DQ array for the input data for any pixel (or affected neighboring pixels) identified as a statistical outlier

The outlier detection step serves as a single interface to apply this general process to any JWST data, with specific variations of this algorithm for each type of data. Sub-classes of the outlier detection algorithm have been developed specifically for:

1. Imaging data
2. IFU spectroscopic data
3. TSO data
4. coronagraphic data
5. spectroscopic data

This allows the outlier_detection step to be tuned to the variations in each type of JWST data.

Reference Files

The `outlier_detection` step uses the PARS-OUTLIERDETECTIONSTEP parameter reference file.

PARS-OUTLIERDETECTIONSTEP Parameter Reference File

REFTYPE
PARS-OUTLIERDETECTIONSTEP

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-outlierdetectionstep references based on the following keywords.

Instrument	Keywords
FGS	EXP_TYPE
MIRI	EXP_TYPE, FILTER, SUBARRAY, TSOVISIT
NIRCAM	EXP_TYPE, FILTER, PUPIL, TSOVISIT
NIRISS	EXP_TYPE, FILTER, PUPIL, TSOVISIT

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Step Arguments for Non-IFU data

The `outlier_detection` step for non-IFU data has the following optional arguments that control the behavior of the processing:

`--weight_type` (string, default='exptime')
The type of data weighting to use during resampling; options are 'exptime', 'error', and 'None'.

`--pixfrac` (float, default=1.0)
The pixel fraction used during resampling; valid values go from 0.0 to 1.0.

`--kernel` (string, default='square')
The form of the kernel function used to distribute flux onto a resampled image. Options are 'square', 'turbo', 'point', 'lanczos', and 'tophat'.

`--fillval` (string, default='INDEF')
The value to assign to resampled image pixels that have zero weight or do not receive any flux from any input pixels during drizzling. Any floating-point value, given as a string, is valid. A value of 'INDEF' will use the last zero weight flux.

--nlow (integer, default=0)
The number of low values in each pixel stack to ignore when computing the median value.

--nhigh (integer, default=0)
The number of high values in each pixel stack to ignore when computing the median value.

--maskpt (float, default=0.7)
The percent of maximum weight to use as lower-limit for valid data; valid values go from 0.0 to 1.0.

--snr (string, default='4.0 3.0')
The signal-to-noise values to use for bad pixel identification. Valid values are a pair of floating-point values in a single string.

--scale (string, default='0.5 0.4')
The scaling factor applied to derivative used to identify bad pixels. Valid values are a pair of floating-point values in a single string.

--backg (float, default=0.0)
User-specified background value to apply to the median image.

--save_intermediate_results (boolean, default=False)
Specifies whether or not to save any intermediate products created during step processing.

--resample_data (boolean, default=True)
Specifies whether or not to resample the input images when performing outlier detection.

--good_bits (string, default="~DO_NOT_USE")
The DQ bit values from the input image DQ arrays that should be considered 'good' when building the weight mask. See DQ flag *Parameter Specification* for details.

--scale_detection (bool, default=False)
Specifies whether or not to rescale the individual input images to match total signal when doing comparisons.

--allowed_memory (float, default=None)
Specifies the fractional amount of free memory to allow when creating the resampled image. If `None`, the environment variable `DMODEL_ALLOWED_MEMORY` is used. If not defined, no check is made. If the resampled image would be larger than specified, an `OutputTooLargeError` exception will be generated.

For example, if set to `0.5`, only resampled images that use less than half the available memory can be created.

--in_memory (boolean, default=False)
Specifies whether or not to load and create all images that are used during processing into memory. If `False`, input files are loaded from disk when needed and all intermediate files are stored on disk, rather than in memory.

Step Arguments for IFU data

The `outlier_detection` step for IFU data has the following optional arguments that control the behavior of the processing:

--kernel_size (string, default='7 7')
The size of the kernel to use to normalize the pixel differences. The kernel size must only contain odd values.

--threshold_percent (float, default=99.8)
The threshold (in percent) of the normalized minimum pixel difference used to identify bad pixels. Pixels with a normalized minimum pixel difference above this percentage are flagged as a outlier.

--save_intermediate_results (boolean, default=False)
Specifies whether or not to save any intermediate products created during step processing.

--in_memory (boolean, default=False)
Specifies whether or not to load and create all images that are used during processing into memory. If `False`, input files are loaded from disk when needed and all intermediate files are stored on disk, rather than in memory.

Python Step Design: OutlierDetectionStep

This module provides the sole interface to all methods of performing outlier detection on JWST observations. The `outlier_detection` step supports multiple algorithms and determines the appropriate algorithm for the type of observation being processed. This step supports:

1. **Image modes**: 'FGS_IMAGE', 'MIR_IMAGE', 'NRC_IMAGE', 'NIS_IMAGE'
2. **Spectroscopic modes**: 'MIR_LRS-FIXEDSLIT', 'NRS_FIXEDSLIT', 'NRS_MSASPEC'
3. **Time-Series-Observation(TSO) Spectroscopic modes**: 'MIR_LRS-SLITLESS', 'NRC_TSGRISM', 'NIS_SOSS', 'NRS_BRIGHTOBJ'
4. **IFU Spectroscopic modes**: 'MIR_MRS', 'NRS_IFU'
5. **TSO Image modes**: 'NRC_TSIMAGE'
6. **Coronagraphic Image modes**: 'MIR_LYOT', 'MIR_4QPM', 'NRC_CORON'

This step uses the following logic to apply the appropriate algorithm to the input data:

1. Interpret inputs (ASN table, ModelContainer or CubeModel) to identify all input observations to be processed
2. Read in type of exposures in input by interpreting `meta.exposure.type` from inputs
3. Read in parameters set by user
4. Select outlier detection algorithm based on exposure type
   - **Images**: like those taken with NIRCam, will use *`OutlierDetection`* as described in *Default Outlier Detection Algorithm*
   - **Coronagraphic observations**: use *`OutlierDetection`* with resampling turned off as described in *Default Outlier Detection Algorithm*
   - **Time-Series Observations(TSO)**: both imaging and spectroscopic modes, use *`OutlierDetection`* with resampling turned off as described in *Default Outlier Detection Algorithm*
   - **IFU observations**: use *`OutlierDetectionIFU`* as described in *Outlier Detection for IFU Data*
   - **Long-slit spectroscopic observations**: use *`OutlierDetectionSpec`* as described in *Outlier Detection for Slit-like Spectroscopic Data*
5. Instantiate and run outlier detection class determined for the exposure type using parameter values interpreted from inputs.
6. Return input models with DQ arrays updated with flags for identified outliers

jwst.outlier_detection.outlier_detection_step Module

Public common step definition for OutlierDetection processing.

Classes

OutlierDetectionStep([name, parent, ...])	Flag outlier bad pixels and cosmic rays in DQ array of each input image.

OutlierDetectionStep

class jwst.outlier_detection.outlier_detection_step.**OutlierDetectionStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Flag outlier bad pixels and cosmic rays in DQ array of each input image.

Input images can be listed in an input association file or already opened with a ModelContainer. DQ arrays are modified in place.

Parameters

input_data (*asn file or ModelContainer*) – Single filename association table, or a datamodels.ModelContainer.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

check_input()	Use this method to determine whether input is valid or not.
process(input_data)	Perform outlier detection processing on input data.

Attributes Documentation

class_alias = 'outlier_detection'

spec

```
weight_type = option('ivm','exptime',default='ivm')
pixfrac = float(default=1.0)
kernel = string(default='square') # drizzle kernel
fillval = string(default='INDEF')
nlow = integer(default=0)
nhigh = integer(default=0)
maskpt = float(default=0.7)
grow = integer(default=1)
snr = string(default='5.0 4.0')
scale = string(default='1.2 0.7')
backg = float(default=0.0)
kernel_size = string(default='7 7')
threshold_percent = float(default=99.8)
ifu_second_check = boolean(default=False)
save_intermediate_results = boolean(default=False)
resample_data = boolean(default=True)
good_bits = string(default="~DO_NOT_USE")  # DQ flags to allow
scale_detection = boolean(default=False)
search_output_file = boolean(default=False)
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image
in_memory = boolean(default=False)
```

Methods Documentation

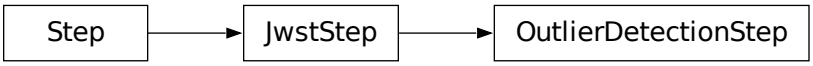
check_input()

Use this method to determine whether input is valid or not.

process(*input_data*)

Perform outlier detection processing on input data.

Class Inheritance Diagram



Default Outlier Detection Algorithm

This module serves as the interface for applying `outlier_detection` to direct image observations, like those taken with MIRI, NIRCam and NIRISS. The code implements the basic outlier detection algorithm used with HST data, as adapted to JWST.

Specifically, this routine performs the following operations:

1. Extract parameter settings from input model and merge them with any user-provided values. See *outlier detection arguments* for the full list of parameters.
2. Convert input data, as needed, to make sure it is in a format that can be processed.
   - A `ModelContainer` serves as the basic format for all processing performed by this step, as each entry will be treated as an element of a stack of images to be processed to identify bad-pixels/cosmic-rays and other artifacts.
   - If the input data is a `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.dat convert it into a ModelContainer. This allows each plane of the cube to be treated as a separate 2D image for resampling (if done) and for combining into a median image.
3. By default, resample all input images.
   - The resampling step starts by computing an output WCS that is large enoug to encompass all the input images.
   - All images from the *same exposure* will get resampled onto this output WCS to create a mosaic of all the chips for that exposure. This product is referred to as a "grouped mosaic" since it groups all the chips from the same exposure into a single image.
   - Each dither position will result in a separate grouped mosaic, so only a single exposure ever contributes to each pixel in these mosaics.
   - An explanation of how all NIRCam multiple detector group mosaics are defined from a single exposure or from a dithered set of exposures (https://jwst-docs.stsci.edu/near-infrared-camera/nircam-operations/nircam-dithers-and-mosaics) can be found here.

- The `fillval` parameter specifies what value to use in the ouptut resampled image for any pixel which has no valid contribution from any input exposure. The default value of `INDEF` indicates that the value from the last exposure will be used, while a value of 0 would result in holes.
- The resampling can be controlled with the `pixfrac`, `kernel` and `weight_type` parameters.
- The `pixfrac` indicates the fraction by which input pixels are "shrunk" before being drizzled onto the output image grid, given as a real number between 0 and 1. This specifies the size of the footprint, or "dropsize", of a pixel in units of the input pixel size.
- The `kernel` specifies the form of the kernel function used to distribute flux onto the separate output images.
- The `weight_type` indicates the type of weighting image to apply with the bad pixel mask. Available options are `ivm` (default) for computing and using an inverse-variance map and `exptime` for weighting by the exposure time.
- The `good_bits` parameter specifies what DQ values from the input exposure should be used when resampling to create the output mosaic. Any pixel with a DQ value not included in this value (or list of values) will be ignored when resampling.
- Resampled images will be written out to disk as `_outlier_i2d.fits` by default.
- **If resampling is turned off** through the use of the `resample_data` parameter, a copy of the unrectified input images (as a ModelContainer) will be used for subsequent processing.

4. Create a median image from all grouped observation mosaics.
   - The median image is created by combining all grouped mosaic images or non-resampled input data (as planes in a ModelContainer) pixel-by-pixel.
   - The `nlow` and `nhigh` parameters specify how many low and high values to ignore when computing the median for any given pixel.
   - The `maskpt` parameter sets the percentage of the weight image values to use, and any pixel with a weight below this value gets flagged as "bad" and ignored when resampled.
   - The median image is written out to disk as `_<asn_id>_median.fits` by default.
5. By default, the median image is blotted back (inverse of resampling) to match each original input image.
   - Blotted images are written out to disk as `_<asn_id>_blot.fits` by default.
   - **If resampling is turned off**, the median image is compared directly to each input image.
6. Perform statistical comparison between blotted image and original image to identify outliers.
   - This comparison uses the original input images, the blotted median image, and the derivative of the blotted image to create a cosmic ray mask for each input image.
   - The derivative of the blotted image gets created using the blotted median image to compute the absolute value of the difference between each pixel and its four surrounding neighbors with the largest value being the recorded derivative.
   - These derivative images are used to flag cosmic rays and other blemishes, such as satellite trails. Where the difference is larger than can be explained by noise statistics, the flattening effect of taking the median, or an error in the shift (the latter two effects are estimated using the image derivative), the suspect pixel is masked.
   - The `backg` parameter specifies a user-provided value to be used as the background estimate. This gets added to the background-subtracted blotted image to attempt to match the original background levels of the original input mosaic so that cosmic-rays (bad pixels) from the input mosaic can be identified more easily as outliers compared to the blotted mosaic.

- Cosmic rays are flagged using the following rule:

$$|image\_input - image\_blotted| > scale * image\_deriv + SNR * noise$$

- The `scale` is defined as the multiplicative factor applied to the derivative which is used to determine if the difference between the data image and the blotted image is large enough to require masking.
- The `noise` is calculated using a combination of the detector read noise and the poisson noise of the blotted median image plus the sky background.
- The user must specify two cut-off signal-to-noise values using the `snr` parameter for determining whether a pixel should be masked: the first for detecting the primary cosmic ray, and the second for masking lower-level bad pixels adjacent to those found in the first pass. Since cosmic rays often extend across several pixels, the adjacent pixels make use of a slightly lower SNR threshold.

7. Update input data model DQ arrays with mask of detected outliers.

Memory Model for Outlier Detection Algorithm

The outlier detection algorithm can end up using massive amounts of memory depending on the number of inputs, the size of each input, and the size of the final output product. Specifically,

1. The input `ModelContainer` or `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jv for IFU data, by default, all input exposures would have been kept open in memory to make processing more efficient.
2. The initial resample step creates an output product for EACH input that is the same size as the final output product, which for imaging modes can span all chips in the detector while also accounting for all dithers. For some Level 3 products, each resampled image can be on the order of 2Gb or more.
3. The median combination step then needs to have all pixels at the same position on the sky in memory in order to perform the median computation. The simplest implementation for this step requires keeping all resampled outputs fully in memory at the same time.

Many Level 3 products only include a modest number of input exposures that can be processed using less than 32Gb of memory at a time. However, there are a number of ways this memory limit can be exceeded. This has been addressed by implementing an overall memory model for the outlier detection that includes options to minimize the memory usage at the expense of file I/O. The control over this memory model happens with the use of the `in_memory` parameter. The full impact of this parameter during processing includes:

1. The `save_open` parameter gets set to `False` (https://docs.python.org/3/library/constants.html#False) when opening the input `ModelContainer` object. This forces all input models in the input `ModelContainer` or `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeModel) to get written out to disk. The ModelContainer then uses the filename of the input model during subsequent processing.
2. The `in_memory` parameter gets passed to the *`ResampleStep`* to set whether or not to keep the resampled images in memory or not. By default, the outlier detection processing sets this parameter to `False` (https://docs.python.org/3/library/constants.html#False) so that each resampled image gets written out to disk.
3. Computing the median image works section-by-section by only keeping 1Mb of each input in memory at a time. As a result, only the final output product array for the final median image along with a stack of 1Mb image sections are kept in memory.
4. The final resampling step also avoids keeping all inputs in memory by only reading each input into memory 1 at a time as it gets resampled onto the final output product.

These changes result in a minimum amount of memory usage during processing at the obvious expense of reading and writing the products from disk.

Outlier Detection for TSO data

Time-series observations (TSO) result in input data stored as a 3D CubeModel where each plane in the cube represents a separate integration without changing the pointing. Normal imaging data benefit from combining all integrations into a single image. TSO data's value, however, comes from looking for variations from one integration to the next. The outlier detection algorithm, therefore, gets run with a few variations to accomodate the nature of these 3D data.

1. Input data is converted from a CubeModel (3D data array) to a ModelContainer
   - Each plane in the original input CubeModel gets copied to a separate model in the ModelContainer
2. The median image is created without resampling the input data
   - All integrations are aligned already, so no resampling or shifting needs to be performed
3. A matched median gets created by combining the single median frame with the noise model for each input integration.
4. Perform statistical comparison between the matched median with each input integration.
5. Update input data model DQ arrays with the mask of detected outliers.

Note: This same set of steps also gets used to perform outlier detection on coronographic data, because it too is processed as 3D (per-integration) cubes.

Outlier Detection for IFU data

Integral Field Unit (IFU) data is handled as 2D images, similar to direct imaging modes. The nature of the detection algorithm, however, is quite different and involves measuring the differences between neighboring pixels in the spatial (cross-dispersion) direction within the IFU slice images. See the *IFU outlier detection* documentation for all the details.

jwst.outlier_detection.outlier_detection Module

Primary code for performing outlier detection on JWST observations.

Functions

flag_cr(sci_image, blot_image[, snr, scale, ...])	Masks outliers in science image by updating DQ in-place
abs_deriv(array)	Take the absolute derivate of a numpy array.

flag_cr

jwst.outlier_detection.outlier_detection.**flag_cr**(*sci_image*, *blot_image*, *snr='5.0 4.0'*, *scale='1.2 0.7'*, *backg=0*, *resample_data=True*, ***kwargs*)

Masks outliers in science image by updating DQ in-place

Mask blemishes in dithered data by comparing a science image with a model image and the derivative of the model image.

Parameters

- **sci_image** (*ImageModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jws – the science data
- **blot_image** (*ImageModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jw – the blotted median image of the dithered science frames
- **snr** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Signal-to-noise ratio
- **scale** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – scaling factor applied to the derivative
- **backg** (*float* (https://docs.python.org/3/library/functions.html#float)) – Background value (scalar) to subtract
- **resample_data** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Boolean to indicate whether blot_image is created from resampled, dithered data or not

abs_deriv

jwst.outlier_detection.outlier_detection.**abs_deriv**(*array*)

Take the absolute derivate of a numpy array.

Classes

OutlierDetection(input_models[, reffiles])	Main class for performing outlier detection.

OutlierDetection

class jwst.outlier_detection.outlier_detection.**OutlierDetection**(*input_models*, *reffiles=None*, ***pars*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Main class for performing outlier detection.

This is the controlling routine for the outlier detection process. It loads and sets the various input data and parameters needed by the various functions and then controls the operation of this process through all the steps used for the detection.

Notes

This routine performs the following operations:

```
1. Extracts parameter settings from input model and merges
   them with any user-provided values
2. Resamples all input images into grouped observation mosaics.
3. Creates a median image from all grouped observation mosaics.
4. Blot median image to match each original input image.
5. Perform statistical comparison between blotted image and original
   image to identify outliers.
6. Updates input data model DQ arrays with mask of detected outliers.
```

Initialize the class with input ModelContainers.

Parameters

- **input_models** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of DataModels*, *str* (https://docs.python.org/3/library/stdtypes.html#str)) – list of data models as ModelContainer or ASN file, one data model for each input image
- **pars** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict), *optional*) – Optional user-specified parameters to modify how outlier_detection will operate. Valid parameters include: - resample_suffix

Attributes Summary

default_suffix	

Methods Summary

blot_median(median_model)	Blot resampled median image back to the detector images.
build_suffix(**pars)	Build suffix.
create_median(resampled_models)	Create a median image from the singly resampled images.
detect_outliers(blot_models)	Flag DQ array for cosmic rays in input images.
do_detection()	Flag outlier pixels in DQ of input images.

Attributes Documentation

default_suffix = 'i2d'

Methods Documentation

blot_median(*median_model*)

Blot resampled median image back to the detector images.

build_suffix(***pars*)

Build suffix.

Class-specific method for defining the resample_suffix attribute using a suffix specific to the sub-class.

create_median(*resampled_models*)

Create a median image from the singly resampled images.

Notes

This version is simplified from astrodrizzle's version in the following ways: - type of combination: fixed to 'median' - 'minmed' not implemented as an option

detect_outliers(*blot_models*)

Flag DQ array for cosmic rays in input images.

The science frame in each ImageModel in input_models is compared to the corresponding blotted median image in blot_models. The result is an updated DQ array in each ImageModel in input_models.

Parameters

- **input_models** (*JWST ModelContainer object*) – data model container holding science ImageModels, modified in place
- **blot_models** (*JWST ModelContainer object*) – data model container holding ImageModels of the median output frame blotted back to the wcs and frame of the ImageModels in input_models

Returns
The dq array in each input model is modified in place

Return type
None

do_detection()

Flag outlier pixels in DQ of input images.

Class Inheritance Diagram

Outlier Detection for IFU Data

This module serves as the interface for applying `outlier_detection` to IFU observations, like those taken with NIRSpec and MIRI. The code implements the basic outlier detection algorithm searching for pixels that are consistent outliers in the calibrated images created by the *calwebb_spec2* pipeline. After launch it was discovered the bad pixels on the MIRI detectors vary with time. The pixels varied from usable to unusable, and at times, back to usable on a time frame that was too short (sometimes as short as 2 days) to fold into the bad pixel mask applied in the *calwebb_detector1* pipeline. At this time it is believed that NIRSpec IFU data also have bad pixels that vary with time, though the time variation is still under study.

An algorithm was developed to flag pixels that are outliers when compared to their neighbors for a set of input files contained in an association. The neighbor pixel differences are the neighbors in spatial direction. For MIRI data ,the neighbor differences are found to the left and right of every science pixel. While for NIRSpec data neighbor differences are found between the pixels above and below every science pixel. The pixel differences for each input model in the association is determined and is stored in a stack of pixel differences. For each pixel the minimum difference through

this stack is determined and normalized. The normalization uses a local median of the difference array (set by the kernel size). A pixel is flagged as an outlier if this normalized minimum difference is greater than the input threshold percentage. Pixels that are found to be outliers are flaged in in the DQ array.

jwst.outlier_detection.outlier_detection_ifu Module

Class definition for performing outlier detection on IFU data.

Classes

OutlierDetectionIFU(input_models[, reffiles])	Sub-class defined for performing outlier detection on IFU data.

OutlierDetectionIFU

class jwst.outlier_detection.outlier_detection_ifu.**OutlierDetectionIFU**(*input_models*, *reffiles=None*, ***pars*)

Bases: *OutlierDetection*

Sub-class defined for performing outlier detection on IFU data.

This is the controlling routine for the outlier detection process. It loads and sets the various input data and parameters needed to flag outliers. Pixel are flagged as outliers based on the MINIMUM difference a pixel has with its neighbor across all the input cal files.

Notes

This routine performs the following operations:

```
1. Extracts parameter settings from input ModelContainer and merges
   them with any user-provided values
2. Loop over cal files
   a. read in science data
   b. Store computed neighbor differences for all the pixels.
      The neighbor pixel  differences are defined by the dispersion axis.
      For MIRI (disp axis = 1) the neighbors to find differences  are to the left␣
↪and right of pixel
      For NIRSpec (disp axis = 0) the neighbors to find the differences are above␣
↪and below the pixel
3. For each input file store the  minimum of the pixel neighbor differences
4. Comparing all the differences from all the input data find the minimum neighbor␣
↪difference
5. Normalize minimum difference to local median of difference array
6. select outliers by flagging those normailzed minimum values > threshold_percent
7. Updates input ImageModel DQ arrays with mask of detected outliers.
```

Initialize class for IFU data processing.

Parameters

- **input_models** (*ModelContainer, str* (https://docs.python.org/3/library/stdtypes.html#str)) – list of data models as ModelContainer or ASN file, one data model for each input 2-D ImageModel
- **reffiles** (dict of JwstDataModel (https://stdatamodels.readthedocs.io/en/latest/api/stdatamodels.jwst.datamodels – Dictionary of datamodels. Keys are reffile_types.

Methods Summary

create_optional_results_model(opt_info)	Creates an OutlierOutputModel from the computed arrays from outlier detection on IFU data.
do_detection()	Split data by detector to find outliers.
flag_outliers(idet, uq_det, ndet_files, ...)	Flag outlier pixels on IFU.

Methods Documentation

create_optional_results_model(*opt_info*)

Creates an OutlierOutputModel from the computed arrays from outlier detection on IFU data.

Parameter

input_model: ~stdatamodels.jwst.datamodels.RampModel

opt_info: tuple The output arrays needed for the OultierOutputModel.

returns
opt_model – The optional OutlierIFUOutputModel to be returned from the outlier_detection_ifu step.

rtype
OutlierIFUOutputModel

do_detection()

Split data by detector to find outliers.

flag_outliers(*idet*, *uq_det*, *ndet_files*, *diffaxis*, *nx*, *ny*, *kern_size*, *threshold_percent*, *save_intermediate_results*, *ifu_second_check*)

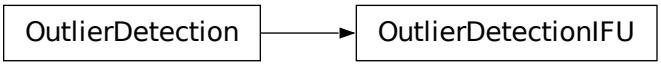
Flag outlier pixels on IFU. In general we are searching for pixels that are a form of a bad pixel but not in bad pixel mask, because the bad pixels vary with time. This program will flag the DQ of input images as DO_NOT_USE and OUTLIER and set the associated science pixel to a Nan. This routine only works on data from one detector.

Parameters

- **idet** (*int* (https://docs.python.org/3/library/functions.html#int)) – Integer indicating which detector we are working with
- **uq_det** (*string array*) – Array of (unique) detector names found input data
- **n_det_files** (*int* (https://docs.python.org/3/library/functions.html#int)) – Number of files for the detector we are working on
- **diffaxis** (*int* (https://docs.python.org/3/library/functions.html#int)) – The axis to form the adjacent pixel differences

- **nx** (*int* (https://docs.python.org/3/library/functions.html#int)) – Size of input data on x axis
- **ny** (*int* (https://docs.python.org/3/library/functions.html#int)) – Since of inut data on y axis
- **threshold_percent** (*float* (https://docs.python.org/3/library/functions.html#float)) – Percent for flagging outliers. Flags pixels where the minimum difference between adjacent pixels for all the input data for a detector is above this percentage. The percentage is based on using all the pixels except a 4 X 4 row and column region around the detector that is often noisy.
- **save_intermediate_results** (*boolean*) – If True then save intermediate output data
- **ifu_second_check** (*boolean*) – If True then perform a secondary check searching for outliers. This will set outliers where ever the difference array of adjacent pixels is a Nan.

Class Inheritance Diagram



Outlier Detection for Slit-like Spectroscopic Data

This module serves as the interface for applying `outlier_detection` to slit-like spectroscopic observations. The code implements the basic outlier detection algorithm used with HST data, as adapted to JWST spectroscopic observations.

Specifically, this routine performs the following operations (modified from the *Default Outlier Detection Algorithm*):

1. Extract parameter settings from input model and merge them with any user-provided values
   - the same set of parameters available to: ref:`Default Outlier Detection Algorithm` also applies to this code
2. Convert input data, as needed, to make sure it is in a format that can be processed
   - A `ModelContainer` serves as the basic format for all processing performed by this step, as each entry will be treated as an element of a stack of images to be processed to identify bad pixels, cosmic-rays and other artifacts
   - If the input data is a `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.dat convert it into a `ModelContainer`. This allows each plane of the cube to be treated as a separate 2D image for resampling (if done) and for combining into a median image.
3. Resample all input images into a `ModelContainer` using `ResampleSpecData`
   - Resampled images are written out to disk if the `save_intermediate_results` parameter is set to `True` (https://docs.python.org/3/library/constants.html#True)
   - **If resampling is turned off**, the original unrectified inputs are used to create the median image for cosmic-ray detection

4. Create a median image from (possibly) resampled `ModelContainer`
   - The median image is written out to disk if the `save_intermediate_results` parameter is set to `True` (https://docs.python.org/3/library/constants.html#True)
5. Blot median image to match each original input image
   - Resampled/blotted images are written out to disk if the `save_intermediate_results` parameter is set to `True` (https://docs.python.org/3/library/constants.html#True)
   - **If resampling is turned off**, the median image is used for comparison with the original input models for detecting outliers
6. Perform statistical comparison between blotted image and original image to identify outliers
7. Update input data model DQ arrays with mask of detected outliers

jwst.outlier_detection.outlier_detection_spec Module

Class definition for performing outlier detection on spectra.

Classes

OutlierDetectionSpec(input_models[, reffiles])	Class definition for performing outlier detection on spectra.

OutlierDetectionSpec

class jwst.outlier_detection.outlier_detection_spec.**OutlierDetectionSpec**(*input_models*, *reffiles=None*, ***pars*)

Bases: *OutlierDetection*

Class definition for performing outlier detection on spectra.

This is the controlling routine for the outlier detection process. It loads and sets the various input data and parameters needed by the various functions and then controls the operation of this process through all the steps used for the detection.

Notes

This routine performs the following operations:

```
1. Extracts parameter settings from input model and merges
   them with any user-provided values
2. Resamples all input images into grouped observation mosaics.
3. Creates a median image from all grouped observation mosaics.
4. Blot median image to match each original input image.
5. Perform statistical comparison between blotted image and original
   image to identify outliers.
6. Updates input data model DQ arrays with mask of detected outliers.
```

Initialize class with input_models.

Parameters

- **input_models** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of DataModels*, *str* (https://docs.python.org/3/library/stdtypes.html#str)) – list of data models as ModelContainer or ASN file, one data model for each input image
- **reffiles** (dict of stdatamodels.jwst.datamodels.JwstDataModel (https://stdatamodels.readthedocs.io/en/latest/api/stdatamodels.jwst.datamodels.JwstDataModel.html#stdatamodels – Dictionary of datamodels. Keys are reffile_types.
- **pars** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict), *optional*) – Optional user-specified parameters to modify how outlier_detection will operate. Valid parameters include: - resample_suffix

Attributes Summary

default_suffix	

Methods Summary

do_detection()	Flag outlier pixels in DQ of input images.

Attributes Documentation

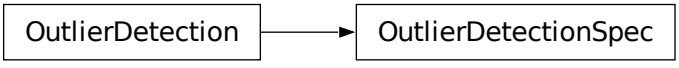
default_suffix = 's2d'

Methods Documentation

do_detection()

Flag outlier pixels in DQ of input images.

Class Inheritance Diagram



jwst.outlier_detection Package

Classes

OutlierDetectionStep([name, parent, ...])	Flag outlier bad pixels and cosmic rays in DQ array of each input image.
OutlierDetectionScaledStep([name, parent, ...])	Flag outlier bad pixels and cosmic rays in DQ array of each input image.
OutlierDetectionStackStep([name, parent, ...])	Class definition for stacked outlier detection.

OutlierDetectionStep

class jwst.outlier_detection.**OutlierDetectionStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Flag outlier bad pixels and cosmic rays in DQ array of each input image.

Input images can be listed in an input association file or already opened with a ModelContainer. DQ arrays are modified in place.

Parameters

input_data (*asn file or ModelContainer*) – Single filename association table, or a datamodels.ModelContainer.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

check_input()	Use this method to determine whether input is valid or not.
process(input_data)	Perform outlier detection processing on input data.

Attributes Documentation

class_alias = 'outlier_detection'

spec

```
weight_type = option('ivm','exptime',default='ivm')
pixfrac = float(default=1.0)
kernel = string(default='square') # drizzle kernel
fillval = string(default='INDEF')
nlow = integer(default=0)
nhigh = integer(default=0)
maskpt = float(default=0.7)
grow = integer(default=1)
snr = string(default='5.0 4.0')
scale = string(default='1.2 0.7')
backg = float(default=0.0)
kernel_size = string(default='7 7')
threshold_percent = float(default=99.8)
ifu_second_check = boolean(default=False)
save_intermediate_results = boolean(default=False)
resample_data = boolean(default=True)
good_bits = string(default="~DO_NOT_USE")  # DQ flags to allow
scale_detection = boolean(default=False)
search_output_file = boolean(default=False)
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image
in_memory = boolean(default=False)
```

Methods Documentation

check_input()

Use this method to determine whether input is valid or not.

process(*input_data*)

Perform outlier detection processing on input data.

OutlierDetectionScaledStep

class jwst.outlier_detection.**OutlierDetectionScaledStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Flag outlier bad pixels and cosmic rays in DQ array of each input image.

Input images can listed in an input association file or already opened with a ModelContainer. DQ arrays are modified in place.

Parameters

input (*asn file or ModelContainer*) – Single filename association table, or a datamodels.ModelContainer.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	Step interface to running outlier_detection.

Attributes Documentation

class_alias = 'outlier_detection_scaled'

spec

```
weight_type = option('ivm','exptime',default='ivm')
pixfrac = float(default=1.0)
kernel = string(default='square') # drizzle kernel
fillval = string(default='INDEF')
nlow = integer(default=0)
nhigh = integer(default=0)
maskpt = float(default=0.7)
snr = string(default='4.0 3.0')
scale = string(default='0.5 0.4')
backg = float(default=0.0)
save_intermediate_results = boolean(default=False)
good_bits = string(default="~DO_NOT_USE")  # DQ flags to allow
```

Methods Documentation

process(*input*)

Step interface to running outlier_detection.

OutlierDetectionStackStep

class jwst.outlier_detection.**OutlierDetectionStackStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

Class definition for stacked outlier detection.

Flag outlier bad pixels and cosmic rays in the DQ array of each input image of a stack of exposures, which in the case of TSO data are from the same data cube.

Input images can listed in an input association file or already opened with a ModelContainer.

DQ arrays are modified in place.

By default, resampling has been disabled. The 'resample_data' attribute can be reset to 'True' to turn on resampling if desired for the data.

Parameters

input (*asn file or ModelContainer*) – Single filename association table, or a datamodels.ModelContainer.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.

- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	Step interface for performing outlier_detection processing.

Attributes Documentation

class_alias = 'outlier_detection_stack'

spec

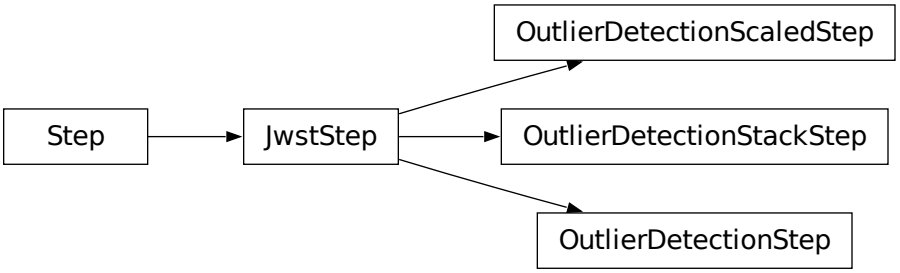
```
weight_type = option('ivm','exptime',default='ivm')
pixfrac = float(default=1.0)
kernel = string(default='square') # drizzle kernel
fillval = string(default='INDEF')
nlow = integer(default=0)
nhigh = integer(default=0)
maskpt = float(default=0.7)
snr = string(default='4.0 3.0')
scale = string(default='0.5 0.4')
backg = float(default=0.0)
save_intermediate_results = boolean(default=False)
resample_data = boolean(default=False)
good_bits = string(default="~DO_NOT_USE")  # DQ flags to allow
```

Methods Documentation

process(*input*)

Step interface for performing outlier_detection processing.

Class Inheritance Diagram



15.1.41 Pathloss Correction

Description

Class
`jwst.pathloss.PathlossStep`

Alias
pathloss

Overview

The `pathloss` step calculates and applies corrections that are needed to account for various types of signal loss in spectroscopic data. The motivation behind the correction is different for the MIRI, NIRSpec, and NIRISS observing modes. For MIRI LRS fixed slit, this correction simply accounts for light not passing through the aperture. For NIRSpec, this correction accounts aperture losses as well as losses in the optical system due to light being scattered outside the grating. For NIRISS SOSS data it corrects for the light that falls outside the detector subarray in use.

Background

The correction is applicable to MIRI LRS fixed-slit, NIRSpec IFU, MSA, and FIXEDSLIT, and NIRISS SOSS data. The description of how the NIRSpec reference files were created and how they are to be applied to NIRSpec data is given in ESA-JWST-SCI-NRS-TN-2016-004 (P. Ferruit: The correction of path losses for uniform and point sources). The NIRISS algorithm was provided by Kevin Volk.

Algorithm

NIRSpec

This step calculates a 1-D correction array as a function of wavelength by interpolating in the pathloss reference file cube at the position of a point source target. It creates 2 pairs of 1-D arrays, a wavelength array (calculated from the WCS applied to the index of the plane in the wavelength direction) and a pathloss correction array calculated by interpolating each plane of the pathloss cube at the position of the source (which is taken from the datamodel). Pairs of these arrays are computed for both point source and uniform source data types. For the uniform source pathloss calculation, there is no dependence on position in the aperture/slit.

Once the 1-D correction arrays have been computed, both forms of the correction (point and uniform) are interpolated, as a function of wavelength, into the 2-D space of the slit or IFU data and attached to the output data model (extensions "PATHLOSS_PS" and "PATHLOSS_UN") as a record of what was computed. The form of the 2-D correction (point or uniform) that's appropriate for the data is divided into the SCI and ERR arrays and propagated into the variance arrays of the science data.

The MSA reference file contains 2 entries: one for a 1x1 slit and one for a 1x3 slit. Each entry contains the pathloss correction for point source and uniform sources. The former depends on the position of the target in the fiducial shutter and wavelength, whereas the latter depends on wavelength only. The point source entry consists of a 3-d array, where 2 of the dimensions map to the location of the source (ranging from -0.5 to 0.5 in both X and Y), while the third dimension carries the wavelength dependence. The 1x3 shutter is 3 times as large in Y as in X.

The entry to use for a point source target is determined by looking at the shutter_state attribute of the slit used. This is a string with a length equal to the number of shutters that make up the slit, with 1 denoting an open shutter, 0 a closed shutter and x the fiducial (target) shutter. The reference entry is determined by how many shutters next to the fiducial shutter are open:

If both adjacent shutters are closed, the 1x1 entry is used. A matching shutter_state might be 'x' or '10x01'

If both adjacent shutters are open, the center region of the 1x3 entry is used. This would be the case for a slit with shutter state '1x1' or '1011x1'.

If one adjacent shutter is open and one closed, the 1x3 entry is used. If the shutter below the fiducial is open and the shutter above closed, then the upper region of the 1x3 pathloss array is used. This is implemented by adding 1 to the Y coordinate of the target position (bringing it into the range +0.5 to +1.5), moving it to the upper third of the pathloss array. A matching shutter state might be '1x' or '11x011'

Similarly, if the shutter below the fiducial is closed and that above is open, the lower third of the pathloss array is used by subtracting 1 from the Y coordinate of the target position (bringing it into the range -1.5 to -0.5). A matching shutter state could be 'x111' or '110x1'.

Once the X and Y coordinates of the source are mapped into a pixel location in the spatial dimensions of the pathloss array using the WCS of the transformation of position to pixel location, the wavelength dependence is determined by interpolating at that (fractional) pixel position in each wavelength plane, resulting in a pair of 1-d arrays of pathloss correction and wavelength. These arrays are used to interpolate the correction for each pixel of the 2-d extracted science array, since each pixel has a different wavelength, and the correction is applied to the science pixel array.

For uniform sources, there is no dependence of the pathloss correction on position, so the correction arrays are just 1-d arrays of correction and wavelength. The correction depends only on the number of shutters in the slit:

If there is 1 shutter, the 1x1 entry is used

If there are 3 or more shutters, the 1x3 entry is used

If there are 2 shutters, the correction used is the average of the 1x1 and 1x3 entries.

Like for the point source case, the 1-d arrays of pathloss correction and wavelength are used to interpolate the correction for each pixel in the science data, using the wavelength of each pixel to interpolate into the pathloss correction array.

MIRI LRS

The algorithm for MIRI LRS mode is largely the same as that for NIRSpec described above, with the exception of the format in which the reference data are stored. First, the position of the target on the detector is estimated from the target RA/Dec given in the exposure header (TARG_RA, TARG_DEC keywords). This position is then used to interpolate within the pathloss reference data to compute a 1-D pathloss correction array. The 1-D pathloss correction is then interpolated into the 2-D space of the data being corrected based on the wavelengths of each pixel in the science data. The 2-D correction array is then applied to the science data and stored (as a record of what was applied) in the output datamodel ("PATHLOSS_PS" extension).

If for any reason the source is determined to be outside of the slit, the correction defaults to the center of the slit.

The MIRI LRS correction is only applicable to point source data. The step is skipped if the SRCTYPE of the input data does not indicate a point source.

NIRISS SOSS

The correction depends on column number in the science data and on the Pupil Wheel position (keyword PWCPOS). It is provided in the reference file as a FITS image of 3 dimensions (to be compatible with the NIRSpec reference file format). The first dimension is a dummy, while the second gives the dependence with row number, and the third with Pupil Wheel position. For the SUBSTEP96 subarray, the reference file data has shape (1, 2040, 17).

The algorithm calculates the correction for each column by simply interpolating along the Pupil Wheel position dimension of the reference file using linear interpolation. The 1-D vector of correction vs. column number is interpolated, as a function of wavelength, into the 2-D space of the science image and divided into the SCI and ERR arrays and propagated into the variance arrays. The 2-D correction array is also attached to the datamodel (extension "PATHLOSS_PS") as a record of what was applied.

Error Propagation

As described above, the NIRSpec and NIRISS correction factors are divided into the SCI and ERR arrays of the science data, and the square of the correction is divided into the variance arrays (VAR_RNOISE, VAR_POISSON, VAR_FLAT) if they exist. For MIRI LRS, the correction factors are multiplicative, hence they are multiplied into the SCI and ERR arrays, and the square of the correction is multiplied into the variance arrays.

Step Arguments

The `pathloss` step has the following optional arguments to control the behavior of the processing.

`--inverse` (boolean, default=False)
A flag to indicate whether the math operations used to apply the flat-field should be inverted (i.e. multiply the pathloss into the science data, instead of the usual division).

`--source_type` (string, default=None)
Force the processing to use the given source type (POINT, EXTENDED), instead of using the information contained in the input data. Only applicable to NIRSpec data.

`--user_slit_loc` (float, default=None)
Only applies to MIRI LRS fixed-slit exposures. Offset the target location along the dispersion direction of the slit by this amount, in units of arcsec. By definition, the center of the slit is at 0, and the edges in the dispersion direction are about +/-0.255 arcsec.

Reference File

The `pathloss` correction step uses a PATHLOSS reference file.

PATHLOSS Reference File

REFTYPE
PATHLOSS

Data model
`PathlossModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.PathlossModel.html#jwst.datamodels.Pa
`MirLrsPathlossModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirLrsPathlossModel.html#jwst

The PATHLOSS reference file contains correction factors as functions of source position in the aperture and wavelength.

Reference Selection Keywords for PATHLOSS

CRDS selects appropriate PATHLOSS references based on the following keywords. PATHLOSS is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for PATHLOSS

In addition to the standard reference file keywords listed above, the following keywords are *required* in PATHLOSS reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for PATHLOSS*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

Reference File Format

The PATHLOSS reference files are FITS files with extensions for each of the aperture types. The FITS primary HDU does not contain a data array.

MIRI LRS Fixed Slit

The MIRI LRS Fixed Slit reference file has the following FITS structure:

HDU	EXTNAME	XTENSION	NAXIS	Dimensions
1	PATHLOSS	BinTable	2	3 columns x 388 rows

The PATHLOSS extension contains wavelength, pathloss correction, and pathloss uncertainty data. The table has the following format:

Column Name	Data Type	Units	Dimensions
WAVELENGTH	float32	microns	388
PATHLOSS	float32	N/A	388 x 50 x 20
PATHLOSS_ERR	float32	N/A	388 x 50 x 20

The pathloss data in each table row are stored as a 2-D array, containing correction factors as a function of source position (relative to the LRS slit reference point) in the spatial and spectral directions, respectively. Wavelength dependence runs along the row axis of the table. The correction factors are multiplicative and hence get multiplied into the science data being corrected.

NIRSpec IFU

The NIRSpec IFU PATHLOSS reference file just four extensions: one pair for point sources and one pair for uniform sources. In each pair, there are either 3-D arrays for point sources, because the pathloss correction depends on the position of the source in the aperture, or 1-D arrays for uniform sources. The pair of arrays are the pathloss correction itself as a function of decenter in the aperture (pointsource only) and wavelength, and the variance on this measurement (currently estimated). The data apply equally to all IFU slices. The structure of the FITS file is as follows:

HDU	EXTNAME	XTENSION	Dimensions	Data type
1	PS	ImageHDU	21 x 21 x 21	float64
2	PSVAR	ImageHDU	21 x 21 x 21	float64
3	UNI	ImageHDU	21	float64
4	UNIVAR	ImageHDU	21	float64

NIRSpec Fixed Slit

The NIRSpec Fixed Slit reference file has the following FITS structure:

HDU	EXTNAME	EXTVER	XTENSION	Dimensions	Data type
1	PS	1	ImageHDU	21 x 21 x 21	float64
2	PSVAR	1	ImageHDU	21 x 21 x 21	float64
3	UNI	1	ImageHDU	21	float64
4	UNIVAR	1	ImageHDU	21	float64
5	PS	2	ImageHDU	21 x 21 x 21	float64
6	PSVAR	2	ImageHDU	21 x 21 x 21	float64
7	UNI	2	ImageHDU	21	float64
8	UNIVAR	2	ImageHDU	21	float64
9	PS	3	ImageHDU	21 x 21 x 21	float64
10	PSVAR	3	ImageHDU	21 x 21 x 21	float64
11	UNI	3	ImageHDU	21	float64
12	UNIVAR	3	ImageHDU	21	float64
13	PS	4	ImageHDU	21 x 21 x 21	float64
14	PSVAR	4	ImageHDU	21 x 21 x 21	float64
15	UNI	4	ImageHDU	21	float64
16	UNIVAR	4	ImageHDU	21	float64

HDU's 1–4 are for the S200A1 aperture, 5–8 are for S200A2, 9–12 are for S200B1, and 13–16 are for S1600A1. Currently there is no reference data for the S400A1 aperture.

NIRSpec MSASPEC

The NIRSpec MSASPEC reference file has 2 sets of 4 extensions: one set for the 1x1 aperture (slitlet) size and one set for the 1x3 aperture (slitlet) size. Currently there is not any reference data for other aperture sizes. The FITS file has the following structure:

HDU	EXTNAME	EXTVER	XTENSION	Dimensions	Data type
1	PS	1	ImageHDU	21 x 63 x 21	float64
2	PSVAR	1	ImageHDU	21 x 63 x 21	float64
3	UNI	1	ImageHDU	21	float64
4	UNIVAR	1	ImageHDU	21	float64
5	PS	2	ImageHDU	21 x 63 x 21	float64
6	PSVAR	2	ImageHDU	21 x 63 x 21	float64
7	UNI	2	ImageHDU	21	float64
8	UNIVAR	2	ImageHDU	21	float64

NIRISS SOSS

The NIRISS SOSS reference file has just 1 extension HDU. The structure of the FITS file is as follows:

HDU	EXTNAME	XTENSION	Dimensions	Data type
1	PS	ImageHDU	17 x 2040 x 1	float32

The PS extension contains a 3-D array of correction values. The third dimension (length = 1) is a dummy to force the array dimensionality to be the same as the NIRSpec reference file arrays. The other 2 dimensions refer to the number of columns in the correction (the same as the number of columns in the science data) and the range of values for the Pupil Wheel position (PWCPOS).

WCS Header Keywords

The headers of the pathloss extensions in all of the above reference files should contain WCS information that describes what variables the correction depends on and how they relate to the dimensions of the correction array.

NIRSpec

For the NIRSpec reference files (IFU, Fixed Slit, and MSASPEC), the WCS keywords should have the values shown in the tables below. Dimension 1 expresses the decenter along the dispersion direction for a point source:

Keyword	Value	Comment
CRPIX1	1.0	Reference pixel in fastest dimension
CRVAL1	-0.5	Coordinate value at this reference pixel
CDELT1	0.05	Change in coordinate value for unit change in index
CTYPE1	'UNITLESS'	Type of physical coordinate in this dimension

Dimension 2 expresses the decenter along the direction perpendicular to the dispersion for a point source:

CRPIX2	1.0	Reference pixel in fastest dimension
CRVAL2	-0.5	Coordinate value at this reference pixel
CDELT2	0.05	Change in coordinate value for unit change in index
CTYPE2	'UNITLESS'	Type of physical coordinate in this dimension

Dimension 3 expresses the change of correction as a function of wavelength:

CRPIX3	1.0	Reference pixel in fastest dimension
CRVAL3	6.0E-7	Coordinate value at this reference pixel
CDELT3	2.35E-7	Change in coordinate value for unit change in index
CTYPE3	'METER'	Type of physical coordinate in this dimension

NIRISS SOSS

The NIRISS SOSS reference file should also have WCS components, but their interpretation is different from those in the NIRSpec reference file. Dimension 1 expresses the column number in the science data:

Keyword	Value	Comment
CRPIX1	5.0	Reference pixel in fastest dimension
CRVAL1	5.0	Coordinate value at this reference pixel
CDELT1	1.0	Change in coordinate value for unit change in index
CTYPE1	'PIXEL'	Type of physical coordinate in this dimension

Dimension 2 expresses the value of the PWCPOS keyword:

CRPIX2	9.0	Reference pixel in fastest dimension
CRVAL2	245.304	Coordinate value at this reference pixel
CDELT2	0.1	Change in coordinate value for unit change in index
CTYPE2	'Pupil Wheel Setting'	Type of physical coordinate in this dimension

Dimension 3 is a dummy axis for the NIRISS SOSS reference file:

CRPIX3	1.0	Reference pixel in fastest dimension
CRVAL3	1.0	Coordinate value at this reference pixel
CDELT3	1.0	Change in coordinate value for unit change in index
CTYPE3	'Dummy'	Type of physical coordinate in this dimension

MIRI LRS

For the MIRI LRS reference file, the WCS keywords should have the values shown in the tables below. Dimension 1 expresses the decenter of a point source in the spatial direction (perpendicular to dispersion):

Keyword	Value	Comment
CRPIX1	1.0	Reference pixel in the spatial direction
CRVAL1	-25.0	Coordinate value at this reference pixel
CDELT1	1.0	Change in coordinate value for unit change in index
CTYPE1	'UNITLESS'	Type of physical coordinate in this dimension

Dimension 2 expresses the decenter along the dispersion for a point source:

CRPIX2	1.0	Reference pixel in dispersion direction
CRVAL2	-5.0	Coordinate value at this reference pixel
CDELT2	0.5	Change in coordinate value for unit change in index
CTYPE2	'UNITLESS'	Type of physical coordinate in this dimension

Note that WCS keywords related to wavelength are not needed for the MIRI LRS reference file, because an array of wavelength values is included in the table of reference data.

jwst.pathloss Package

Classes

PathLossStep([name, parent, config_file, ...])	PathLossStep: Apply the path loss correction to a science exposure.

PathLossStep

class jwst.pathloss.**PathLossStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

PathLossStep: Apply the path loss correction to a science exposure.

Pathloss depends on the centering of the source in the aperture if the source is a point source.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'pathloss'

reference_file_types = ['pathloss']

spec

```
inverse = boolean(default=False)     # Invert the operation
source_type = string(default=None)  # Process as specified source type
user_slit_loc = float(default=None)   # User-provided correction to MIRI LRS␣
↪source location
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.42 Persistence Correction

Description

Class
jwst.persistence.PersistenceStep

Alias
persistence

Based on a model, this step computes the number of traps that are expected to have captured or released a charge during an exposure. The released charge is proportional to the persistence signal, and this will be subtracted (group by group) from the science data. An image of the number of filled traps at the end of the exposure will be written as an output file, in order to be used as input for correcting the persistence of a subsequent exposure.

There may be an input traps-filled file (defaults to 0), giving the number of traps that are filled in each pixel. There is one plane of this 3-D image for each "trap family," sets of traps having similar capture and decay parameters. The traps-filled file is therefore coupled with the trappars reference table, which gives parameters family-by-family. There are currently three trap families.

If an input traps-filled file was specified, the contents of that file will be updated (decreased) to account for trap decays from the EXPEND of the traps-filled file to the EXPSTART of the current science file before starting the processing of the science data.

When processing a science image, the traps-filled file is the basis for computing the number of trap decays, which are computed group-by-group. On the other hand, the trap-density file is the basis for predicting trap captures, which are computed at the end of each integration. The traps-filled file will be updated (decreased by the number of traps that released a charge) after processing each group of the science image. The traps-filled file will then be increased by the number of traps that were predicted to have captured a charge by the end of each integration.

There is often a reset at the beginning of each integration, and if so, that time (a frame time) will be included in the trap capture for each integration, and it will be included for the tray decay for the first group of each integration.

The number of trap decays in a given time interval is computed as follows:

$$n\_decays = trapsfilled \cdot (1 - exp(-\Delta t/\tau))$$

where trapsfilled is the number of filled traps, i.e. the value of the traps-filled image at the beginning of the time interval, for the current trap family and at the current pixel; Δt is the time interval (seconds) over which the decay is computed; and τ is the reciprocal of the absolute value of the decay parameter (column name "decay_param") for the current trap family. Since this is called for each group, the value of the traps-filled image must be updated at the end of each group.

For each pixel, the persistence in a group is the sum of the trap decays over all trap families. This persistence is subtracted from the science data for the current group. Pixels that have large persistence values subtracted from them are flagged in the DQ array, as information to the user (see the Step Arguments section).

Trap capture is more involved than is trap decay. The computation of trap capture is different for an impulse (e.g. a cosmic-ray event) than for a ramp, and saturation also affects capture. Computing trap capture needs an estimate of the ramp slope, and it needs the locations (pixel number and group number) of cosmic-ray jumps. At the time of writing, the `persistence` step is run before the `jump` step, so the GROUPDQ array in the input to `persistence` does not contain the information that is required to account for cosmic-ray events.

Because the `persistence` step is run before `ramp_fit`, the persistence step does not have the value of the slope, so the step must compute its own estimate of the slope. The algorithm is as follows. First of all, the slope must be computed before the loop over groups in which trap decay is computed and persistence is corrected, since that correction will in general change the slope. Within an integration, the difference is taken between groups of the ramp. The difference is set to a very large value if a group is saturated. (The "very large value" is the larger of 10^5 and twice the maximum difference between groups.) The difference array is then sorted. All the differences affected by saturation will be at the high end. Cosmic-ray affected differences should be just below, except for jumps that are smaller than some of the noise. We can then ignore saturated values and jumps by knowing how many of them there are (which we know from the GROUPDQ array). The average of the remaining differences is the slope. The slope is needed with two different units. The `grp_slope` is the slope in units of DN (data numbers) per group. The `slope` is in units of (DN / persistence saturation limit) / second, where "persistence saturation limit" is the (pixel-dependent) value (in DN) from the PERSAT reference file.

The number of traps that capture charge is computed at the end of each integration. The number of captures is computed in three phases: the portion of the ramp that is increasing smoothly from group to group; the saturated portion (if any) of the ramp; the contribution from cosmic-ray events.

For the smoothly increasing portion of the ramp, the time interval over which traps capture charge is nominally $nresets \cdot tframe + ngroups \cdot tgroup$ where nresets is the number of resets at the beginning of the integration, tframe is the frame time, and tgroup is the group time. However, this time must be reduced by the group time multiplied by the number of groups for which the data value exceeds the persistence saturation limit. This reduced value is $Deltat$ in the expression below.

The number of captures in each pixel during the integration is:

$$trapsfilled = 2 \cdot (trapdensity \cdot slope^2 \cdot (\Delta t^2 \cdot (par0 + par2)/2 + par0 \cdot (\Delta t \cdot \tau + \tau^2) \cdot exp(-\Delta t/\tau) - par0 \cdot \tau^2))$$

where par0 and par2 are the values from columns "capture0" and "capture2" respectively, from the trappars reference table, and τ is the reciprocal of the absolute value from column "capture1", for the row corresponding to the current trap family. trapdensity is the relative density of traps, normalized to a median of 1. Δt is the time interval in seconds over which the charge capture is to be computed, as described above. slope is the ramp slope (computed before the loop over groups), in units of fraction of the persistence saturation limit per second. This returns the number of traps that were predicted to be filled during the integration, due to the smoothly increasing portion of the ramp. This is passed as input to the function that computes the additional traps that were filled due to the saturated portion of the ramp.

"Saturation" in this context means that the data value in a group exceeds the persistence saturation limit, i.e. the value in the PERSAT reference file. filled_during_integration is (initially) the array of the number of pixels that were filled, as returned by the function for the smoothly increasing portion of the ramp. In the function for computing decays for the saturated part of the ramp, for pixels that are saturated in the first group, filled_during_integration is set to $trapdensity \cdot par2$ (column "capture2"). This accounts for "instantaneous" traps, ones that fill over a negligible time scale.

The number of "exponential" traps (as opposed to instantaneous) is:

$$exp\_filled\_traps = filled\_during\_integration - trapdensity \cdot par2$$

and the number of traps that were empty and could be filled is:

$$empty\_traps = trapdensity \cdot par0 - exp\_filled\_traps$$

so the traps that are filled depending on the exponential component is:

$$new\_filled\_traps = empty\_traps \cdot (1 - exp(-sattime/\tau))$$

where sattime is the duration in seconds over which the pixel was saturated.

Therefore, the total number of traps filled during the current integration is:

$$filled\_traps = filled\_during\_integration + new\_filled\_traps$$

This value is passed to the function that computes the additional traps that were filled due to cosmic-ray events.

The number of traps that will be filled due to a cosmic-ray event depends on the amount of time from the CR event to the end of the integration. Thus, we must first find (via the flags in the GROUPDQ extension) which groups and which pixels were affected by CR hits. This is handled by looping over group number, starting with the second group (since we currently don't flag CRs in the first group), and selecting all pixels with a jump. For these pixels, the amplitude of the jump is computed to be the difference between the current and previous groups minus grp_slope (the slope in DN per group). If a jump is negative, it will be set to zero.

If there was a cosmic-ray hit in group number k, then

$$\Delta t = (ngroups - k - 0.5) \cdot tgroup$$

is the time from the CR-affected group to the end of the integration, with the approximation that the CR event was in the middle (timewise) of the group. The number of traps filled as a result of this CR hit is:

$$crfilled = 2 \cdot trapdensity \cdot jump \cdot (par0 \cdot (1 - exp(-\Delta t/\tau)) + par2)$$

and the number of filled traps for the current pixel will be incremented by that amount.

Input

The input science file is a RampModel.

A trapsfilled file (TrapsFilledModel) may optionally be passed as input as well. This normally would be specified unless the previous exposure with the current detector was taken more than several hours previously, that is, so long ago that persistence from that exposure could be ignored. If none is provided, an array filled with 0 will be used as the starting point for computing new traps-filled information.

Output

The output science file is a RampModel, a persistence-corrected copy of the input data.

A second output file will be written, with suffix "_trapsfilled". This is a TrapsFilledModel, the number of filled traps at each pixel at the end of the exposure. This takes into account the capture of charge by traps due to the current science exposure, as well as the release of charge from traps given in the input trapsfilled file, if one was specified. Note that this file will always be written, even if no input_trapsfilled file was specified. This file should be passed as input to the next run of the persistence step for data that used the same detector as the current run. Pass this file using the input_trapsfilled argument.

If the user specifies `save_persistence=True`, a third output file will be written, with suffix "_output_pers". This is a RampModel matching the output science file, but this gives the persistence that was subtracted from each group in each integration.

Step Arguments

The persistence step has three step-specific arguments.

- `--input_trapsfilled`

`input_trapsfilled` is the name of the most recent trapsfilled file for the current detector. If this is not specified, an array of zeros will be used as an initial value. If this is specified, it will be used to predict persistence for the input science file. The step writes an output trapsfilled file, and that could be used as input to the persistence step for a subsequent exposure.

- `--flag_pers_cutoff`

If this floating-point value is specified, pixels that receive a persistence correction greater than or equal to `flag_pers_cutoff` DN (the default is 40) are flagged in the PIXELDQ array of the output file with the DQ value "PERSISTENCE".

- `--save_persistence`

If this boolean parameter is specified and is True (the default is False), the persistence that was subtracted (group by group, integration by integration) will be written to an output file with suffix "_output_pers".

Reference Files

The `persistence` step uses *TRAPDENSITY*, *PERSAT*, and *TRAPPARS* reference files.

TRAPDENSITY Reference File

REFTYPE
TRAPDENSITY

Data model
`TrapDensityModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.TrapDensityModel.html#jwst.datam

The TRAPDENSITY reference file contains a pixel-by-pixel map of the trap density.

Reference Selection Keywords for TRAPDENSITY

CRDS selects appropriate TRAPDENSITY references based on the following keywords. TRAPDENSITY is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for TRAPDENSITY

In addition to the standard reference file keywords listed above, the following keywords are *required* in TRAPDENSITY reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for TRAPDENSITY*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector

Reference File Format

TRAPDENSITY reference files are FITS format, with 2 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	int
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

PERSAT Reference File

REFTYPE
PERSAT

Data model
`PersistenceSatModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.PersistenceSatModel.html#jwst.

The PERSAT reference file contains a pixel-by-pixel map of the persistence saturation (full well) threshold.

Reference Selection Keywords for PERSAT

CRDS selects appropriate PERSAT references based on the following keywords. PERSAT is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for PERSAT

In addition to the standard reference file keywords listed above, the following keywords are *required* in PERSAT reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for PERSAT*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector

Reference File Format

PERSAT reference files are FITS format, with 2 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	int
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

TRAPPARS Reference File

REFTYPE
TRAPPARS

Data model
`TrapParsModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.TrapParsModel.html#jwst.datamodels.T

The TRAPPARS reference file contains default parameter values used in the persistence correction.

Reference Selection Keywords for TRAPPARS

CRDS selects appropriate TRAPPARS references based on the following keywords. TRAPPARS is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for TRAPPARS

In addition to the standard reference file keywords listed above, the following keywords are *required* in TRAPPARS reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for TRAPPARS*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector

Reference File Format

TRAPPARS reference files are FITS format, with 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	Dimensions
TRAPPARS	BINTABLE	TFIELDS = 4

The format and contents of the table extension is as follows:

Column name	Data type	Description
capture0	float	Coefficient of exponential capture term
capture1	float	Minus the reciprocal of capture e-folding time
capture2	float	The "instantaneous" capture coefficient
decay_param	float	Minus the reciprocal of decay e-folding time

At the present time, there are no persistence reference files for MIRI and NIRSpec. CRDS will return "N/A" for the names of the reference files if the persistence step is run on MIRI or NIRSpec data, in which case the input will be returned unchanged, except that the primary header keyword S_PERSIS will will have been set to 'SKIPPED'.

jwst.persistence Package

Classes

PersistenceStep([name, parent, config_file, ...])	PersistenceStep: Correct a science image for persistence.

PersistenceStep

class jwst.persistence.**PersistenceStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

PersistenceStep: Correct a science image for persistence.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'persistence'

reference_file_types = ['trapdensity', 'trappars', 'persat']

spec

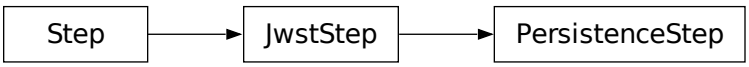
```
input_trapsfilled = string(default="") # Name of the most recent trapsfilled␣
↪file for the current detector
flag_pers_cutoff = float(default=40.) # Pixels with persistence correction >=␣
↪this value in DN will be flagged in the DQ
save_persistence = boolean(default=False) # Save subtracted persistence to an␣
↪output file with suffix '_output_pers'
save_trapsfilled = boolean(default=True) # Save updated trapsfilled file with␣
↪suffix '_trapsfilled'
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.43 Photometric Calibration

Description

Class

jwst.photom.PhotomStep

Alias

photom

The `photom` step applies flux (photometric) calibrations to a data product to convert the data from units of countrate to surface brightness (or, in some cases described below, to units of flux density). The calibration information is read from a photometric reference file. The exact nature of the calibration information loaded from the reference file and applied to the science data depends on the instrument mode, as described below.

This step relies on having wavelength information available when working on spectroscopic data and therefore the *assign_wcs* step *must* be applied before executing the `photom` step. Pixels with wavelengths that are outside of the range covered by the calibration reference data are set to zero and flagged in the DQ array as "DO_NOT_USE." Some spectroscopic modes also rely on knowing whether the target is a point or extended source and therefore the *srctype* step *must* be applied before executing the `photom` step.

Upon successful completion of this step, the status keyword S_PHOTOM will be set to "COMPLETE". Furthermore, the BUNIT keyword value in the SCI and ERR extension headers of the science product are updated to reflect the change in units.

Imaging and non-IFU Spectroscopy

Photom Data

For these instrument modes the PHOTOM reference file contains a table of exposure parameters that define various instrument configurations and the flux conversion data for each of those configurations. The table contains one row for each allowed combination of exposure parameters, such as detector, filter, pupil, and grating. The photom step searches the table for the row that matches the parameters of the science exposure and then loads the calibration information from that row of the table. Note that for NIRSpec fixed-slit mode, the step will search the table for each slit in use in the exposure, using the table row that corresponds to each slit.

For these table-based PHOTOM reference files, the calibration information in each row includes a scalar flux conversion constant, as well as optional arrays of wavelength and relative response (as a function of wavelength). For spectroscopic data, if the photom step finds that the wavelength and relative response arrays in the reference table row are populated, it loads those 1-D arrays and interpolates the response values into the 2-D space of the science image based on the wavelength at each pixel.

For NIRSpec spectroscopic and NIRISS SOSS data, the conversion factors in the PHOTOM reference file give results in flux density (MJy). For point sources, the output of the photom step will be in these units. For extended sources, however, the values will be divided by the average solid angle of a pixel to give results in surface brightness (MJy/sr). The photom step determines whether the target is a point or extended source from the SRCTYPE keyword value, which is set by the *srctype* step. If the SRCTYPE keyword is not present or is set to "UNKNOWN", the default behavior is to treat it as a uniform/extended source.

The combination of the scalar conversion factor and the 2-D response values are then applied to the science data, including the SCI and ERR arrays, as well as the variance (VAR_POISSON, VAR_RNOISE, and VAR_FLAT) arrays. The correction values are multiplied into the SCI and ERR arrays, and the square of the correction values are multiplied into the variance arrays.

The scalar conversion constant is copied to the header keyword PHOTMJSR, which gives the conversion from DN/s to megaJy/steradian (or to megajanskys, for NIRSpec and NIRISS SOSS point sources, as described above) that was applied to the data. The step also computes the equivalent conversion factor to units of microJy/square-arcsecond (or microjanskys) and stores it in the header keyword PHOTUJA2.

MIRI Imaging

For MIRI imaging mode, the reference file can optionally contain a table of coefficients that are used to apply time-dependent corrections to the scalar conversion factor. If the time-dependent coefficients are present in the reference file, the photom step will apply the correction based on the observation date of the exposure being processed.

NIRSpec Fixed-Slit Primary Slit

The primary slit in a NIRSpec fixed-slit exposure receives special handling. If the primary slit, as given by the "FXD_SLIT" keyword value, contains a point source, as given by the "SRCTYPE" keyword, it is necessary to know the photometric conversion factors for both a point source and a uniform source for use later in the *master background* step in Stage 3 processing. The point source version of the photometric correction is applied to the slit data, but that correction is not appropriate for the background signal contained in the slit, and hence corrections must be applied later in the *master background* step.

So in this case the `photom` step will compute 2D arrays of conversion factors that are appropriate for a uniform source and for a point source, and store those correction factors in the "PHOTOM_UN" and "PHOTOM_PS" extensions, respectively, of the output data product. The point source correction array is also applied to the slit data.

Note that this special handling is only needed when the slit contains a point source, because in that case corrections to the wavelength grid are applied by the *wavecorr* step to account for any source mis-centering in the slit and the photometric conversion factors are wavelength-dependent. A uniform source does not require wavelength corrections and hence the photometric conversions will differ for point and uniform sources. Any secondary slits that may be included in a fixed-slit exposure do not have source centering information available, so the *wavecorr* step is not applied, and hence there's no difference between the point source and uniform source photometric conversions for those slits.

Pixel Area Data

For all instrument modes other than NIRSpec the photom step loads a 2-D pixel area map when an AREA reference file is available and appends it to the science data product. The pixel area map is copied into an image extension called "AREA" in the science data product.

The step also populates the keywords PIXAR_SR and PIXAR_A2 in the science data product, which give the average pixel area in units of steradians and square arcseconds, respectively. For most instrument modes, the average pixel area values are copied from the primary header of the PHOTOM reference file. For NIRSpec, however, the pixel area values are copied from a binary table extension in the AREA reference file.

NIRSpec IFU

The photom step uses the same type of tabular PHOTOM reference file for NIRSpec IFU exposures as discussed above for other modes, where there is a single table row that corresponds to a given exposure's filter and grating settings. It retrieves the scalar conversion constant, as well as the 1-D wavelength and relative response arrays, from that row. It also loads the IFU pixel area data from the AREA reference file.

It then uses the scalar conversion constant, the 1-D wavelength and relative response, and pixel area data to compute a 2-D sensitivity map (pixel-by-pixel) for the entire science image. The 2-D SCI and ERR arrays in the science exposure are multiplied by the 2D sensitivity map, which converts the science pixels from countrate to surface brightness. Variance arrays are multiplied by the square of the conversion factors.

MIRI MRS

For the MIRI MRS mode, the PHOTOM reference file contains 2-D arrays of sensitivity factors and pixel sizes that are loaded into the step. As with NIRSpec IFU, the sensitivity and pixel size data are used to compute a 2-D sensitivity map (pixel-by-pixel) for the entire science image. This is multiplied into both the SCI and ERR arrays of the science exposure, which converts the pixel values from countrate to surface brightness. Variance arrays are multiplied by the square of the conversion factors.

MIRI MRS data have a time-variable photometric response that is significant at long wavelengths. A correction has been derived from observations of calibration standard stars. The form of the correction uses an exponential function that asymptotically approaches a constant value in each wavelength band. A plot of the count rate loss in each MRS band, as a function of time, is shown in Figure 1.

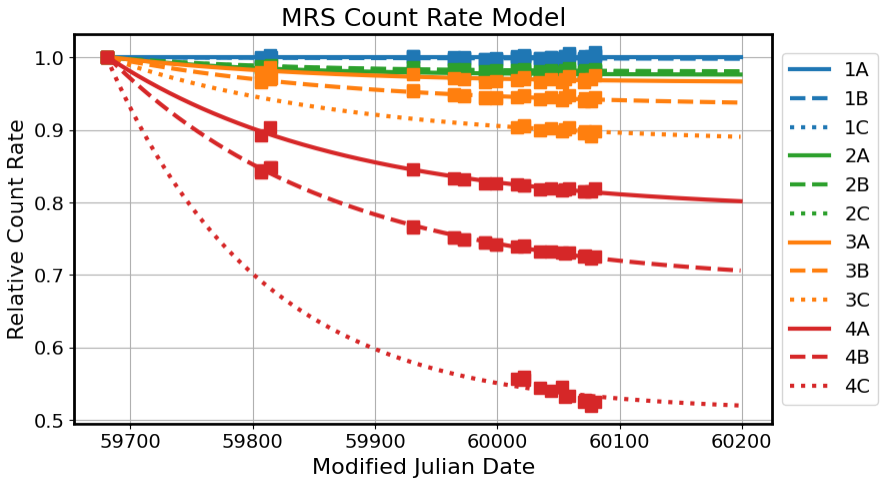


Figure 1: Time-dependent decrease in the observed MRS count rate as measured from internal flat-field exposures. Solid points illustrate measurements at the central wavelength of each of the 12 MRS bands; curves represent the best fit models used for correction in the pipeline.

The MRS photom reference file contains a table of correction coefficients for each band in which a correction has been determined. If the time-dependent coefficients are present in the reference file for a given band, the photom step will apply the correction to the exposure being processed.

Arguments

The `photom` step has the following optional arguments.

`--inverse` (boolean, default=False)
A flag to indicate whether the math operations used to apply the correction should be inverted (i.e. divide the calibration data into the science data, instead of the usual multiplication).

`--source_type` (string, default=None)
Force the processing to use the given source type (POINT, EXTENDED), instead of using the information contained in the input data.

`--mrs_time_correction` (boolean, default=True)
A flag to indicate whether to turn on the time and wavelength dependent correction for MIRI MRS data.

Reference Files

The `photom` step uses *PHOTOM* and pixel *AREA* reference files. The AREA reference file is only used when processing imaging and NIRSpec IFU observations.

PHOTOM Reference File

REFTYPE
PHOTOM

The PHOTOM reference file contains conversion factors for putting pixel values into physical units.

Reference Selection Keywords for PHOTOM

CRDS selects appropriate PHOTOM references based on the following keywords. PHOTOM is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, BAND, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for PHOTOM

In addition to the standard reference file keywords listed above, the following keywords are *required* in PHOTOM reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for PHOTOM*):

Keyword	Data Model Name	Instruments
DETECTOR	model.meta.instrument.detector	FGS, MIRI, NIRCam, NIRISS
EXP_TYPE	model.meta.exposure.type	All
BAND	model.meta.instrument.band	MIRI

Tabular PHOTOM Reference File Format

PHOTOM reference files are FITS format. For all modes except MIRI MRS, the PHOTOM file contains tabular data in a BINTABLE extension with EXTNAME = 'PHOTOM'. The FITS primary HDU does not contain a data array. The contents of the table extension vary a bit for different instrument modes, as shown in the tables below.

Data model
`FgsImgPhotomModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.FgsImgPhotomModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
FGS	Image	photmjsr	float	scalar	MJy/steradian/(DN/sec)
		uncertainty	float	scalar	MJy/steradian/(DN/sec)

Data model
`MirImgPhotomModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirImgPhotomModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	Image	filter	string	12	N/A
		subarray	string	15	N/A
		photmjsr	float	scalar	MJy/steradian/(DN/sec)
		uncertainty	float	scalar	MJy/steradian/(DN/sec)

The MIRI Imager PHOTOM reference file can contain an optional BINTABLE extension named "TIMECOEFF", containing coefficients for an time-dependent correction. The format of this additional table extension is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
TIMECOEFF	BINTABLE	2	TFIELDS = 3	float32

Data model

`MirLrsPhotomModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirLrsPhotomModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	LRS	filter	string	12	N/A
		subarray	string	15	N/A
		photmjsr	float	scalar	MJy/steradian/(DN/sec)
		uncertainty	float	scalar	MJy/steradian/(DN/sec)
		nelem	integer	scalar	N/A
		wavelength	float	array	microns
		relresponse	float	array	unitless
		reluncertainty	float	array	unitless

Data model

`NrcImgPhotomModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrcImgPhotomModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRCam	Image	filter	string	12	N/A
		pupil	string	12	N/A
		photmjsr	float	scalar	MJy/steradian/(DN/sec)
		uncertainty	float	scalar	MJy/steradian/(DN/sec)

Data model

`NrcWfssPhotomModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrcWfssPhotomModel.html#jwst

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRCam	WFSS	filter	string	12	N/A
		pupil	string	15	N/A
		order	integer	scalar	N/A
		photmjsr	float	scalar	MJy/steradian/(DN/sec)
		uncertainty	float	scalar	MJy/steradian/(DN/sec)
		nelem	integer	scalar	N/A
		wavelength	float	array	microns
		relresponse	float	array	unitless
		reluncertainty	float	array	unitless

Data model

NisImgPhotomModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NisImgPhotomModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRISS	Image	filter	string	12	N/A
		pupil	string	12	N/A
		photmjsr	float	scalar	MJy/steradian/(DN/sec)
		uncertainty	float	scalar	MJy/steradian/(DN/sec)

Data model

NisSossPhotomModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NisSossPhotomModel.html#jwst.

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRISS	SOSS	filter	string	12	N/A
		pupil	string	15	N/A
		order	integer	scalar	N/A
		photmj	float	scalar	MJy/(DN/sec)
		uncertainty	float	scalar	MJy/(DN/sec)
		nelem	integer	scalar	N/A
		wavelength	float	array	microns
		relresponse	float	array	unitless
		reluncertainty	float	array	unitless

Data model

NisWfssPhotomModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NisWfssPhotomModel.html#jwst.

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRISS	WFSS	filter	string	12	N/A
		pupil	string	15	N/A
		order	integer	scalar	N/A
		photmjsr	float	scalar	MJy/steradian/(DN/sec)
		uncertainty	float	scalar	MJy/steradian/(DN/sec)
		nelem	integer	scalar	N/A
		wavelength	float	array	microns
		relresponse	float	array	unitless
		reluncertainty	float	array	unitless

Data model

NrsFsPhotomModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrsFsPhotomModel.html#jwst.datam

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	Fixed Slit	filter	string	12	N/A
		grating	string	15	N/A
		slit	string	15	N/A
		photmj	float	scalar	MJy/(DN/sec)
		uncertainty	float	scalar	MJy/(DN/sec)
		nelem	integer	scalar	N/A
		wavelength	float	array	microns
		relresponse	float	array	unitless
		reluncertainty	float	array	unitless

Data model

NrsMosPhotomModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrsMosPhotomModel.html#jwst.d

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	MOS and IFU	filter	string	12	N/A
		grating	string	15	N/A
		photmj	float	scalar	MJy/(DN/sec)
		uncertainty	float	scalar	MJy/(DN/sec)
		nelem	integer	scalar	N/A
		wavelength	float	array	microns
		relresponse	float	array	unitless
		reluncertainty	float	array	unitless

Row Selection

A row of data within the table is selected by the `photom` step based on the optical elements in use for the exposure. The selection attributes are always contained in the first few columns of the table. The remaining columns contain the data needed for photometric conversion. The row selection criteria for each instrument/mode are:

- **FGS:**
  - No selection criteria (table contains a single row)
- **MIRI:**
  - Imager and LRS: Filter and Subarray
  - MRS: Does not use table-based reference file (see below)
- **NIRCam:**
  - All: Filter and Pupil
- **NIRISS:**
  - Imaging: Filter and Pupil
  - Spectroscopic: Filter, Pupil, and Order number
- **NIRSpec:**
  - IFU and MOS: Filter and Grating
  - Fixed Slits: Filter, Grating, and Slit name

Note: For spectroscopic data the Nelem column should be present. Its value must be greater than 0, and Nelem entries are read from each of the Wavelength and Relresponse arrays. Nelem is not used for imaging data because there are no columns containing arrays.

The primary header of the tabular PHOTOM reference files contains the keywords PIXAR_SR and PIXAR_A2, which give the average pixel area in units of steradians and square arcseconds, respectively.

MIRI MRS Photom Reference File Format

Data model

`MirMrsPhotomModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirMrsPhotomModel.html#jwst.da

For MIRI MRS, the PHOTOM file contains 2-D arrays of conversion factors in IMAGE extensions. The FITS primary HDU does not contain a data array. The format and content of the MIRI MRS PHOTOM reference file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	1032 x 1024	float
ERR	IMAGE	2	1032 x 1024	float
DQ	IMAGE	2	1032 x 1024	integer
PIXSIZ	IMAGE	2	1032 x 1024	float
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A
TIMECOEFF_CH1[1]	BINTABLE	2	TFIELDS = 5	N/A
TIMECOEFF_CH2[1]	BINTABLE	2	TFIELDS = 5	N/A
TIMECOEFF_CH3[1]	BINTABLE	2	TFIELDS = 5	N/A
TIMECOEFF_CH4[1]	BINTABLE	2	TFIELDS = 5	N/A

The SCI extension contains a 2D array of inverse sensitivity factors corresponding to each pixel in a 2D MRS slice image. The sensitivity factors are in units of (MJy/pixel)/(DN/sec). The ERR extension contains a 2D array of uncertainties for the SCI values, in the same units. The DQ extension contains a 2D array of bit-encoded data quality flags for the SCI values. The DQ_DEF extension contains a table listing the definitions of the values used in the DQ array. The PIXSIZ extension contains a 2D array of pixel sizes (i.e. solid angles), in units of square-arcsec.

The SCI and PIXSIZ array values are both divided into the science product SCI and ERR arrays, yielding surface brightness in units of mJy/sq-arcsec.

Scalar PHOTMJSR and PHOTUJA2 values are stored in primary header keywords in the MIRI MRS PHOTOM reference files and are copied into the science product header by the photom step.

The TIMECOEFF_CH tables contain the parameters to correct the MRS time-dependent throughput loss. If these tables do not exist in the reference file, then the MIRI MRS time-dependent correction is skipped.

Constructing a PHOTOM Reference File

The most straight-forward way to construct a tabular PHOTOM reference file is to populate a data model within python and then save the data model to a FITS file. Each instrument mode has its own photom data model, as listed above, which contains the columns of information unique to that instrument.

A NIRCam WFSS photom reference file, for example, could be constructed as follows from within the python environment:

```
>>> import numpy as np
>>> from stdatamodels.jwst import datamodels
>>> filter = np.array(['GR150C', 'GR150R'])
>>> pupil = np.array(['F140M', 'F200W'])
>>> order = np.array([1, 1], dtype=np.int16)
>>> photf = np.array([1.e-15, 3.e-15], dtype=np.float32)
>>> uncer = np.array([1.e-17, 3.e-17], dtype=np.float32)
>>> nrows = len(filter)
>>> nx = 437
```

(continues on next page)

[1] Optional extension. If present, the MRS time-dependent throughput correction can be applied.

(continued from previous page)

```
>>> nelem = np.zeros(nrows, dtype=np.int16) + nx
>>> temp_wl = np.linspace(1.0, 5.0, nx, dtype=np.float32).reshape(1, nx)
>>> wave = np.zeros((nrows, nx), np.float32)
>>> wave[:] = temp_wl.copy()
>>> resp = np.ones((nrows, nx), dtype=np.float32)
>>> resp_unc = np.zeros((nrows, nx), dtype=np.float32)
>>> data_list = [(filter[i], pupil[i], order[i], photf[i], uncer[i], nelem[i],
...               wave[i], resp[i], resp_unc[i]) for i in range(nrows)]
>>> data = np.array(data_list,
...                 dtype=[('filter', 'S12'),
...                        ('pupil', 'S15'),
...                        ('order', '<i2'),
...                        ('photmjsr', '<f4'),
...                        ('uncertainty', '<f4'),
...                        ('nelem', '<i2'),
...                        ('wavelength', '<f4', (nx,)),
...                        ('relresponse', '<f4', (nx,)),
...                        ('reluncertainty', '<f4', (nx,))])
>>> output = datamodels.NrcWfssPhotomModel(phot_table=data)
>>> output.save('nircam_photom_0001.fits')
'nircam_photom_0001.fits'
```

AREA Reference File

REFTYPE
AREA

The AREA reference file contains pixel area information for a given instrument mode.

Reference Selection Keywords for AREA

CRDS selects appropriate AREA references based on the following keywords. AREA is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, FILTER, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, FILTER, PUPIL, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, FILTER, PUPIL, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, FILTER, GRATING, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for AREA

In addition to the standard reference file keywords listed above, the following keywords are *required* in AREA reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for AREA*):

Keyword	Data Model Name	Instrument
DETECTOR	model.meta.instrument.detector	All
EXP_TYPE	model.meta.exposure.type	MIRI, NIRCam, NIRISS, NIRSpec
FILTER	model.meta.instrument.filter	MIRI, NIRCam, NIRISS, NIRSpec
PUPIL	model.meta.instrument.pupil	NIRCam, NIRISS
GRATING	model.meta.instrument.grating	NIRSpec

Reference File Format

AREA reference files are FITS format. For imaging modes (FGS, MIRI, NIRCam, and NIRISS) the AREA reference files contain 1 IMAGE extension, while reference files for NIRSpec spectroscopic modes contain 1 BINTABLE extension. The FITS primary HDU does not contain a data array.

Imaging Modes

Data model
`PixelAreaModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.PixelAreaModel.html#jwst.datamodels

The format of imaging mode AREA reference files is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float

The SCI extension data array contains a 2-D pixel-by-pixel map of relative pixel areas, normalized to a value of 1.0. The absolute value of the nominal pixel area is given in the primary header keywords PIXAR_SR and PIXAR_A2, in units of steradians and square arcseconds, respectively. These keywords should have the same values as the PIXAR_SR and PIXAR_A2 keywords in the header of the corresponding PHOTOM reference file.

NIRSpec Fixed-Slit Mode

Data model
`NirspecSlitAreaModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecSlitAreaModel.html#jw

The BINTABLE extension has EXTNAME='AREA' and has column characteristics shown below. There is one row for each of the 5 fixed slits, with `slit_id` values of "S200A1", "S200A2", "S400A1", "S200B1", and "S1600A1". The pixel area values are in units of square arcseconds and represent the nominal area of any pixel illuminated by the slit.

Column name	Data type
slit_id	char*15
pixarea	float

NIRSpec MOS Mode

Data model
`NirspecMosAreaModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecMosAreaModel.html#jw

The BINTABLE extension has EXTNAME='AREA' and has column characteristics shown below. There is one row for each shutter in each MSA quadrant. The quadrant and shutter values are 1-indexed. The pixel area values are in units of square arcseconds and represent the nominal area of any pixel illuminated by a given MSA shutter.

Column name	Data type
quadrant	integer
shutter_x	integer
shutter_y	integer
pixarea	float

NIRSpec IFU Mode

Data model
NirspecIfuAreaModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NirspecIfuAreaModel.html#jwst

The BINTABLE extension has EXTNAME='AREA' and has column characteristics shown below. There is one row for each of the 30 IFU slices, with the `slice_id` values being 0-indexed (i.e. range from 0 to 29). The pixel area values are in units of square arcseconds and represent the nominal area of any pixel illuminated by a given IFU slice.

Column name	Data type
slice_id	integer
pixarea	float

jwst.photom Package

Classes

PhotomStep([name, parent, config_file, ...])	PhotomStep: Module for loading photometric conversion information from

PhotomStep

class jwst.photom.**PhotomStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

PhotomStep: Module for loading photometric conversion information from
reference files and attaching or applying them to the input science data model

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'photom'

reference_file_types = ['photom', 'area']

spec

```
inverse = boolean(default=False)    # Invert the operation
source_type = string(default=None)  # Process as specified source type.
mrs_time_correction = boolean(default=True) # Apply the MIRI MRS time dependent␣
↪correction
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.44 Pipeline Modules

Pipeline Stages

End-to-end calibration of JWST data is divided into 3 main stages of processing:

- Stage 1 consists of detector-level corrections that are performed on a group-by-group basis, followed by ramp fitting. The output of stage 1 processing is a countrate image per exposure, or per integration for some modes. Details of this pipeline can be found at:
  - *calwebb_detector1: Stage 1 Detector Processing*
- Stage 2 processing consists of additional instrument-level and observing-mode corrections and calibrations to produce fully calibrated exposures. The details differ for imaging and spectroscopic exposures, and there are some corrections that are unique to certain instruments or modes. Details are at:
  - *calwebb_image2: Stage 2 Imaging Processing*
  - *calwebb_spec2: Stage 2 Spectroscopic Processing*
- Stage 3 processing consists of routines that work with multiple exposures and in most cases produce some kind of combined product. There are unique pipeline modules for stage 3 processing of imaging, spectroscopic, coronagraphic, AMI, and TSO observations. Details of each are available at:
  - *calwebb_image3: Stage 3 Imaging Processing*
  - *calwebb_spec3: Stage 3 Spectroscopic Processing*
  - *calwebb_coron3: Stage 3 Coronagraphic Processing*
  - *calwebb_ami3: Stage 3 Aperture Masking Interferometry (AMI) Processing*
  - *calwebb_tso3: Stage 3 Time-Series Observation(TSO) Processing*

In addition, there are several pipeline modules designed for special instrument or observing modes, including:

- *calwebb_dark* for processing dark exposures
- *calwebb_guider* for calibrating FGS guide star data
- *calwebb_wfs-image3* for stage 3 WFS&C processing

The table below represents the same information as described above, but alphabetically ordered by pipeline class.

Pipeline Class	Alias	Used For
`Ami3Pipeline`	calwebb_ami3	Stage 3: NIRISS AMI mode
`Coron3Pipeline`	calwebb_coron3	Stage 3: Coronagraphic mode
`DarkPipeline`	calwebb_dark	Stage 1: darks
`Detector1Pipeline`	calwebb_detector1	Stage 1: all modes
`GuiderPipeline`	calwebb_guider	Stage 1+2: FGS guiding modes
`Image2Pipeline`	calwebb_image2	Stage 2: imaging modes
`Image3Pipeline`	calwebb_image3	Stage 3: imaging modes
`Spec2Pipeline`	calwebb_spec2	Stage 2: spectroscopy modes
`Spec3Pipeline`	calwebb_spec3	Stage 3: spectroscopy modes
`Tso3Pipeline`	calwebb_tso3	Stage 3: TSO modes
`WfsCombineStep`	calwebb_wfs-image3	Stage 3: WFS&C imaging

Pipelines vs. Exposure Type

The data from different observing modes needs to be processed with different combinations of the pipeline stages listed above. The proper pipeline selection is usually based solely on the exposure type (EXP_TYPE keyword value). Some modes, however, require additional selection criteria, such as whether the data are to be treated as Time-Series Observations (TSO). Some EXP_TYPEs are exclusively TSO, while others depend on the value of the TSOVISIT keyword. The following table lists the pipeline modules that should get applied to various observing modes, based on these selectors. Exposure types that do not allow TSO mode are marked as "N/A" in the TSOVISIT column.

EXP_TYPE	TSOVISIT	Stage 1 Pipeline	Stage 2 Pipeline	Stage 3 Pipeline
FGS_DARK	N/A	*calwebb_dark*	N/A	N/A
FGS_SKYFLAT FGS_INTFLAT	N/A	*calwebb_detector1*	N/A	N/A
FGS_FOCUS	N/A	*calwebb_detector1*	*calwebb_image2*	N/A
FGS_IMAGE	N/A	*calwebb_detector1*	*calwebb_image2*	*calwebb_image3*
FGS_ID-STACK FGS_ID-IMAGE FGS_ACQ1 FGS_ACQ2 FGS_TRACK FGS_FINEGUID	N/A	*calwebb_guider*	N/A	N/A
MIR_DARKIMG MIR_DARKMRS	N/A	*calwebb_dark*	N/A	N/A

continues on next page

Table 3 – continued from previous page

EXP_TYPE	TSOVISIT	Stage 1 Pipeline	Stage 2 Pipeline	Stage 3 Pipeline
MIR_FLATIMAC MIR_FLATIMAC EXT MIR_FLATMRS MIR_FLATMRS-EXT	N/A	*calwebb_detector1*	N/A	N/A
MIR_TACQ	N/A	*calwebb_detector1*	*calwebb_image2*	N/A
MIR_CORONCA	N/A	*calwebb_detector1*	*calwebb_image2*	N/A
MIR_IMAGE	False	*calwebb_detector1*	*calwebb_image2*	*calwebb_image3*
	True	*calwebb_detector1*	*calwebb_image2*	*calwebb_tso3*
MIR_LRS-FIXEDSLIT	N/A	*calwebb_detector1*	*calwebb_spec2*	*calwebb_spec3*
MIR_LRS-SLITLESS	True	*calwebb_detector1*	*calwebb_spec2*	*calwebb_tso3*
	False	*calwebb_detector1*	*calwebb_spec2*	N/A
MIR_MRS	N/A	*calwebb_detector1*	*calwebb_spec2*	*calwebb_spec3*
MIR_LYOT MIR_4QPM	N/A	*calwebb_detector1*	*calwebb_image2*	*calwebb_coron3*
NRC_DARK	N/A	*calwebb_dark*	N/A	N/A
NRC_FLAT NRC_LED NRC_GRISM	N/A	*calwebb_detector1*	N/A	N/A

continues on next page

Table 3 – continued from previous page

EXP_TYPE	TSOVISIT	Stage 1 Pipeline	Stage 2 Pipeline	Stage 3 Pipeline
NRC_TACQ NRC_TACONFIR NRC_FOCUS	N/A	*calwebb_detector1*	*calwebb_image2*	N/A
NRC_IMAGE	N/A	*calwebb_detector1*	*calwebb_image2*	*calwebb_image3*
NRC_CORON	N/A	*calwebb_detector1*	*calwebb_image2*	*calwebb_coron3*
NRC_WFSS	N/A	*calwebb_detector1*	*calwebb_spec2*	*calwebb_spec3*
NRC_TSIMAGE	True	*calwebb_detector1*	*calwebb_image2*	*calwebb_tso3*
NRC_TSGRISM	True	*calwebb_detector1*	*calwebb_spec2*	*calwebb_tso3*
NIS_DARK	N/A	*calwebb_dark*	N/A	N/A
NIS_LAMP NIS_EXTCAL	N/A	*calwebb_detector1*	N/A	N/A
NIS_TACQ NIS_TACONFIRI NIS_FOCUS	N/A	*calwebb_detector1*	*calwebb_image2*	N/A
NIS_IMAGE	N/A	*calwebb_detector1*	*calwebb_image2*	*calwebb_image3*
NIS_AMI	N/A	*calwebb_detector1*	*calwebb_image2*	calwebb_ami3
NIS_WFSS	N/A	*calwebb_detector1*	*calwebb_spec2*	*calwebb_spec3*
NIS_SOSS	True	*calwebb_detector1*	*calwebb_spec2*	*calwebb_tso3*

continues on next page

Table 3 – continued from previous page

EXP_TYPE	TSOVISIT	Stage 1 Pipeline	Stage 2 Pipeline	Stage 3 Pipeline
	False	*calwebb_detector1*	*calwebb_spec2*	*calwebb_spec3*
NRS_DARK	N/A	*calwebb_dark*	N/A	N/A
NRS_AUTOWAV NRS_AUTOFLA NRS_LAMP	N/A	*calwebb_detector1*	*calwebb_spec2*	N/A
NRS_IMAGE NRS_WATA NRS_MSATA NRS_TACONFIR NRS_CONFIRM NRS_FOCUS NRS_MIMF	N/A	*calwebb_detector1*	*calwebb_image2*	N/A
NRS_FIXEDSLI NRS_IFU NRS_MSASPEC	N/A	*calwebb_detector1*	*calwebb_spec2*	*calwebb_spec3*
NRS_BRIGHTOI	True	*calwebb_detector1*	*calwebb_spec2*	*calwebb_tso3*

Wavefront Sensing and Control Images

Exposures obtained by any instrument for the purpose of WaveFront Sensing and Control (WFS&C) use a dedicated processing flow through the pipeline stages.

- Stage 1: WFS&C exposures use the same *calwebb_detector1* pipeline processing and steps as regular images.
- Stage 2: WFS&C exposures use the same *calwebb_image2* pipeline processing and steps as regular images. CRDS reftype `pars-image2pipeline` has a specific parameter reference for WFS&C processing. The processing is identical with other image processing except for the omission of the *resample* step.
- Stage 3: The ASN generator identifies pairs of dithered WFS&C images to be combined via the "PATTTYPE"

keyword value "WFSC". The resulting ASN uses the *calwebb_wfs-image3* pipeline for stage 3 processing. This pipeline consists of the single step *wfs_combine*.

Configuration File Deprecation

Up to version 1.1.0, the primary way specific pipelines were referred to was by their configuration file name, i.e. `calwebb_detector1.cfg`. These configuration files were delivered as part of the JWST calibration package. Below is the table that matched configuration file to observing mode it was intended to be used with.

Post-1.1.0, configuration files are no longer the primary identifier of pipelines. Instead, pipelines are identified by their full class name, i.e. `jwst.pipeline.Detector1Pipeline`, or by their simple name, or alias, i.e. `calwebb_detector1`. How a pipeline is run is determined by the input data and what parameter reference file in CRDS is selected by that data. The reftype for each pipeline, or step, is determined by appending the class name of the step to the string `pars-`. For example, the reftype for `jwst.pipeline.Detector1Pipeline` is `pars-detector1pipeline`. Which specific reference file for a reftype is then determined by the data, just as with any other reference file.

As a result, there are a few pipelines that no longer exist explicitly by name, because they were only a configuration file for an already existing pipeline class. The pipelines continue to operate correctly for the specific cases, because the parameter references pulled from CRDS will have the correct configuration. The following table lists the deprecated configuration files and what pipeline should now be referred to.

Deprecated CFG	Pipeline Class	Alias
calwebb_nrslamp-spec2.cfg	*jwst.pipeline.Spec2Pipeline*	*calwebb_spec2*
calwebb_tso1.cfg	*jwst.pipeline.Detector1Pipeline*	*calwebb_detector1*
calwebb_tso-image2.cfg	*jwst.pipeline.Image2Pipeline*	*calwebb_image2*
calwebb_tso-spec2.cfg	*jwst.pipeline.Spec2Pipeline*	*calwebb_spec2*
calwebb_wfs-image2.cfg	*jwst.pipeline.Image2Pipeline*	*calwebb_image2*

The deprecated configuration to mode mapping up to version 1.1.0 is in the table below. This table is given only as historical reference for software and documentation that used this terminology.

Pipeline Class	Configuration File	Used For
Detector1Pipeline	calwebb_detector1.cfg	Stage 1: all non-TSO modes
	calwebb_tso1.cfg	Stage 1: all TSO modes
DarkPipeline	calwebb_dark.cfg	Stage 1: darks
GuiderPipeline	calwebb_guider.cfg	Stage 1+2: FGS guiding modes
Image2Pipeline	calwebb_image2.cfg	Stage 2: imaging modes
	calwebb_tso-image2.cfg	Stage 2: TSO imaging modes
	calwebb_wfs-image2.cfg	Stage 2: WFS&C imaging
Spec2Pipeline	calwebb_spec2.cfg	Stage 2: spectroscopy modes
	calwebb_tso-spec2.cfg	Stage 2: TSO spectral modes
	calwebb_nrslamp-spec2.cfg	Stage 2: NIRSpec lamps
Image3Pipeline	calwebb_image3.cfg	Stage 3: imaging modes
WfsCombineStep	calwebb_wfs-image3.cfg	Stage 3: WFS&C imaging
Spec3Pipeline	calwebb_spec3.cfg	Stage 3: spectroscopy modes
Ami3Pipeline	calwebb_ami3.cfg	Stage 3: NIRISS AMI mode
Coron3Pipeline	calwebb_coron3.cfg	Stage 3: Coronagraphic mode
Tso3Pipeline	calwebb_tso3.cfg	Stage 3: TSO modes

calwebb_detector1: Stage 1 Detector Processing

Class
jwst.pipeline.Detector1Pipeline

Alias
calwebb_detector1

The `Detector1Pipeline` applies basic detector-level corrections to all exposure types (imaging, spectroscopic, coronagraphic, etc.). It is applied to one exposure at a time. It is sometimes referred to as "ramps-to-slopes" processing, because the input raw data are in the form of one or more ramps (integrations) containing accumulating counts from the non-destructive detector readouts and the output is a corrected countrate (slope) image.

There are two general configurations for this pipeline, depending on whether the data are to be treated as a Time Series Observation (TSO). The configuration is provided by CRDS and the reftype `pars-detector1pipeline`. In general, for Non-TSO exposures, all applicable steps are applied to the data. For TSO exposures, some steps are set to be skipped by default (see the list of steps in the table below).

The list of steps applied by the `Detector1Pipeline` pipeline is shown in the table below. Note that MIRI exposures use some instrument-specific steps and some of the steps are applied in a different order than for Near-IR (NIR) instrument exposures.

Several steps in this pipeline include special handling for NIRCam "Frame 0" data. The NIRCam instrument has the ability to downlink the image from the initial readout that follows the detector reset at the start of each integration in an exposure. These images are distinct from the first group of each integration when on-board frame averaging is done. In these cases, the first group contains data from multiple frames, while frame zero is always composed of just the first frame following the reset. It can be used to recover an estimated slope for pixels that go into saturation already in the first group (see more details on that process in the *ramp_fitting* step). In order for the frame zero image to be utilized during ramp fitting, it must have all of the same calibrations and corrections applied as the first group in the various `Detector1Pipeline` steps. This includes the *saturation*, *superbias*, *refpix*, and *linearity* steps. Other steps do not have a direct effect on either the first group or frame zero pixel values.

Near-IR Step	Non-TSO	TSO	MIRI Step	Non-TSO	TSO
group_scale	✓	✓	*group_scale*	✓	✓
dq_init	✓	✓	*dq_init*	✓	✓
			emicorr	✓	✓
saturation	✓	✓	*saturation*	✓	✓
ipc[1]			*ipc*		
superbias	✓	✓	*firstframe*	✓	
refpix	✓	✓	*lastframe*	✓	
			reset	✓	✓
linearity	✓	✓	*linearity*	✓	✓
persistence[2]	✓		*rscd*	✓	
dark_current	✓	✓	*dark_current*	✓	✓
			refpix	✓	✓
charge_migration[3]	✓				
jump	✓	✓	*jump*	✓	✓
ramp_fitting	✓	✓	*ramp_fitting*	✓	✓
gain_scale	✓	✓	*gain_scale*	✓	✓

[1] By default, the parameter reference `pars-detector1pipeline` retrieved from CRDS will skip the *ipc* step for all instruments.
[2] The *persistence* step is currently hardwired to be skipped in the `Detector1Pipeline` module for all NIRSpec exposures.
[3] By default, the *charge_migration* step is skipped in the `Detector1Pipeline` module for all instruments.

Arguments

The `calwebb_detector1` pipeline has one optional argument:

```
--save_calibrated_ramp  boolean  default=False
```

If set to `True`, the pipeline will save intermediate data to a file as it exists at the end of the *jump* step. The data at this stage of the pipeline are still in the form of the original 4D ramps (ncols x nrows x ngroups x nints) and have had all of the detector-level correction steps applied to it, including the detection and flagging of Cosmic-Ray (CR) hits within each ramp (integration). If created, the name of the intermediate file will be constructed from the root name of the input file, with the new product type suffix "_ramp" appended, e.g. "jw80600012001_02101_00003_mirimage_ramp.fits".

Inputs

4D raw data

Data model
`RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamodels.RampMo

File suffix
_uncal

The input to `Detector1Pipeline` is a single raw exposure, e.g. "jw80600012001_02101_00003_mirimage_uncal.fits", which contains the original raw data from all of the detector readouts in the exposure (ncols x nrows x ngroups x nintegrations).

Note that in the operational environment, the input will be in the form of a `Level1bModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.Level1bModel.html#jwst.datamodels.Level1bModel), which only contains the 4D array of detector pixel values, along with some optional extensions. When such a file is loaded into the pipeline, it is immediately converted into a `RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamodels.RampModel), and has all additional data arrays for errors and Data Quality flags created and initialized to zero.

The input can also contain a 3D cube of NIRCam "Frame 0" data, where each image plane in the 3D cube is the initial frame for each integration in the exposure. Only present when the option to downlink the frame zero data was selected in the observing program.

Outputs

4D corrected ramp

Data model
`RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamodels.RampMo

File suffix
_ramp

Result of applying all pipeline steps up through the *jump* step, to produce corrected and CR-flagged 4D ramp data, which will have the same data dimensions as the input raw 4D data (ncols x nrows x ngroups x nints). Only created when the pipeline argument `--save_calibrated_ramp` is set to `True` (default is `False`).

2D countrate product

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
or `IFUImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamode

File suffix
_rate

All types of inputs result in a 2D countrate product, based on averaging over all of the integrations within the exposure. The output file will be of type "_rate", e.g. "jw80600012001_02101_00003_mirimage_rate.fits". The 2D "_rate" product is passed along to subsequent pipeline modules for all non-TSO and non-Coronagraphic exposures. For MIRI MRS and NIRSpec IFU exposures, the output data model will be `IFUImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamodels.IFUImageModel), while all others will be `ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamode

3D countrate product

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_rateints

A 3D countrate product is created that contains the individual results of each integration. The 2D countrate images for each integration are stacked along the 3rd axis of the data cubes (ncols x nrows x nints). This output file will be of type "_rateints". The 3D "_rateints" product is passed along to subsequent pipeline modules for all TSO and Coronagraphic exposures.

PARS-DETECTOR1PIPELINE Parameter Reference File

REFTYPE
PARS-DETECTOR1PIPELINE

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-detector1pipeline references based on the following keywords.

Instrument	Keywords
FGS	TSOVISIT
MIRI	TSOVISIT
NIRCAM	TSOVISIT
NIRISS	TSOVISIT
NIRSPEC	TSOVISIT

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

calwebb_image2: Stage 2 Imaging Processing

Class
jwst.pipeline.Image2Pipeline

Alias
calwebb_image2

Stage 2 imaging processing applies additional instrumental corrections and calibrations that result in a fully calibrated individual exposure. There are two parameter references used to control this pipeline, depending on whether the data are to be treated as Time Series Observation (TSO). The parameter reference is provided by CRDS and the reftype `pars-image2pipeline`. In general, for non-TSO exposures, all applicable steps are applied to the data. For TSO exposures, some steps are set to be skipped by default (see the list of steps in the table below).

The list of steps applied by the `Image2Pipeline` pipeline is shown in the table below.

Step	Non-TSO	TSO
background	✓	
assign_wcs	✓	✓
flat_field	✓	✓
photom	✓	✓
resample[1]	✓	

[1] Resampling is only performed for exposure types "MIR_IMAGE", "NRC_IMAGE", and "NIS_IMAGE".

Arguments

The `calwebb_image2` pipeline has one optional argument:

```
--save_bsub  boolean  default=False
```

If set to `True`, the results of the background subtraction step will be saved to an intermediate file, using a product type of "_bsub" or "_bsubints", depending on whether the data are 2D (averaged over integrations) or 3D (per-integration results).

Inputs

2D or 3D countrate data

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
or `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMo

File suffix
_rate or _rateints

The input to `Image2Pipeline` is a countrate exposure, in the form of either "_rate" or "_rateints" data. A single input file can be processed or an ASN file listing multiple inputs can be used, in which case the processing steps will be applied to each input exposure, one at a time. If "_rateints" products are used as input, each step applies its algorithm to each integration in the exposure, where appropriate.

TSO and coronagraphic exposures are expected to use 3D data as input, to be processed on a per-integration basis.

Outputs

2D or 3D background-subtracted data

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
or `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMo

File suffix
_bsub or _bsubints

This is an intermediate product that is only created if "–save_bsub" is set to `True` and will contain the data as output from the *background* step. If the input is a "_rate" product, this will be a "_bsub" product, while "_rateints" inputs will be saved as "_bsubints."

2D or 3D calibrated data

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
or `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMo

File suffix
_cal or _calints

The output is a fully calibrated, but unrectified, exposure, using the product type suffix "_cal" or "_calints", depending on the type of input, e.g. "jw80600012001_02101_00003_mirimage_cal.fits".

2D resampled image

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_i2d

This is the output of the *resample* step and is only created for regular direct imaging observations (not for TSO or coronagraphy 3D data sets). The output file will use the "_i2d" product type suffix, e.g. "jw80600012001_02101_00003_mirimage_i2d.fits". Note that this product is intended for quick-look use only and is not passed along as input to Stage 3 processing. Calibrated, but unrectified (_cal) products are used as input to Stage 3.

PARS-IMAGE2PIPELINE Parameter Reference File

REFTYPE
PARS-IMAGE2PIPELINE

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-image2pipeline references based on the following keywords.

Instrument	Keywords
FGS	EXP_TYPE, TSOVISIT, VISITYPE
MIRI	EXP_TYPE, TSOVISIT, VISITYPE
NIRCAM	EXP_TYPE, TSOVISIT, VISITYPE
NIRISS	EXP_TYPE, TSOVISIT, VISITYPE
NIRSPEC	EXP_TYPE, TSOVISIT, VISITYPE

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

calwebb_spec2: Stage 2 Spectroscopic Processing

Class
jwst.pipeline.Spec2Pipeline

Alias
calwebb_spec2

The `Spec2Pipeline` applies additional instrumental corrections and calibrations to countrate products that result in a fully calibrated individual exposure. There are two general configurations for this pipeline, depending on whether the data are to be treated as Time Series Observation (TSO). In general, for non-TSO exposures, all applicable steps are applied to the data. For TSO exposures, some steps are set to be skipped by default (see the list of steps in the table below).

The `Spec2Pipeline` is the "Swiss army knife" of pipeline modules, containing many steps that are only applied to certain instruments or instrument modes. The logic for determining which steps are appropriate is built into the pipeline module itself and determined by the CRDS `pars-spec2pipeline` parameter reference file. Logic is mostly based on either the instrument name or the exposure type (EXP_TYPE keyword) of the data.

Science Exposures

The list of steps shown in the table below indicates which steps are applied to various spectroscopic modes for JWST science exposures, including TSO exposures. The instrument mode abbreviations used in the table are as follows:

- NIRSpec FS = Fixed Slit
- NIRSpec MOS = Multi-Object Spectroscopy
- NIRSpec IFU = Integral Field Unit
- MIRI FS = LRS Fixed Slit
- MIRI SL = LRS Slitless
- MIRI MRS = Medium Resolution Spectroscopy (IFU)
- NIRISS SOSS = Single Object Slitless Spectroscopy
- NIRISS and NIRCam WFSS = Wide-Field Slitless Spectroscopy

Instrument/Mode	NIRSpec			MIRI			NIRISS		NIRCam	All
Step	FS	MOS	IFU	FS	SL	MRS	SOSS	WFSS	WFSS	TSO
assign_wcs	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
msaflagopen		✓	✓							
nsclean	✓	✓	✓							
imprint		✓	✓							
background	✓	✓	✓	✓		✓	✓	✓	✓	
extract_2d[1]	✓	✓						✓	✓	✓
srctype[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
master_background		✓								
wavecorr	✓	✓								
straylight						✓				
flat_field[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
fringe						✓				
pathloss	✓	✓	✓	✓			✓			
barshadow		✓								
wfss_contam								✓	✓	
photom	✓	✓	✓	✓	✓	✓	✓ [3]	✓	✓	✓
residual_fringe [2]						✓				
pixel_replace [2]	✓	✓	✓	✓	✓			✓	✓	
resample_spec	✓	✓		✓						
cube_build			✓			✓				
extract_1d	✓	✓	✓	✓	✓	✓	✓ [3]	✓	✓	✓

[1]The exact order of the *extract_2d*, *srctype*, and *flat_field* steps depends on the observing mode. For NIRISS and NIRCam WFSS, as well as NIRCam TSO grism exposures, the order is flat_field, extract_2d, and srctype (no wavecorr). For all other modes the order is extract_2d, srctype, wavecorr, and flat_field.

[2]By default this step is skipped in the `calwebb_spec2` pipeline, but is enabled for some modes via overrides provided in parameter reference files.

[3]NIRISS SOSS can have multiple spectral orders contribute flux to one pixel; because photometric correction values depend on the spectral order assigned to a pixel, the order of *photom* and *extract_1d* is swapped for NIRISS SOSS exposures. This allows the ATOCA algorithm to disentangle the spectral orders, such that photometric corrections can be applied to each spectrum separately.

Notice that NIRSpec MOS is the only mode to receive master background subtraction in the `calwebb_spec2` pipeline. All other spectral modes have master background subtraction applied in the *calwebb_spec3* pipeline.

The *resample_spec* step produces a resampled/rectified product for non-IFU modes of some spectroscopic exposures. If the *resample_spec* step is not applied to a given exposure, the *extract_1d* operation will be performed on the original (unresampled) data. The *cube_build* step produces a resampled/rectified cube for IFU exposures, which is then used as input to the *extract_1d* step.

NIRSpec Lamp Exposures

The `Spec2Pipeline` works slightly differently for NIRSpec lamp exposures. These are identified by the EXP_TYPE values of NRS_LAMP, NRS_AUTOWAVE or NRS_AUTOFLAT. Using the EXP_TYPE keyword in this way means that another keyword is needed to specify whether the data are Fixed Slit, MOS, IFU or Brightobj. This is the OPMODE keyword, which maps to the `jwst.datamodel` attribute `.meta.instrument.lamp_mode`. This keyword can take the following values in exposures that undergo `Spec2Pipeline` processing:

- BRIGHTOBJ = Bright Object mode (uses fixed slits)
- FIXEDSLIT = Fixed slit mode
- IFU = Integral Field Unit mode
- MSASPEC = Multi-Object Spectrograph Mode

OPMODE can also take the values of GRATING-ONLY and NONE, but only in some engineering-only situations, and can take the value of IMAGE for imaging data. None of these values will trigger the execution of the `Spec2Pipeline`.

NIRSpec calibration lamps are identified by the LAMP keyword, which maps to the `jwst.datamodel` attribute `.meta.instrument.lamp_state`. The lamps are either line lamps, used for wavelength calibration, or continuum lamps, which are used for flatfielding. Each is paired with a specific grating:

Lamp name	Wavelength range (micron)	Used with grating
FLAT1	1.0 - 1.8	G140M, G140H
FLAT2	1.7 - 3.0	G235M, G235H
FLAT3	2.9 - 5.0	G395M, G395H
FLAT4	0.7 - 1.4	G140M, G140H
FLAT5	1.0 - 5.0	PRISM
LINE1	1.0 - 1.8	G140M, G140H
LINE2	1.7 - 3.0	G235M, G235H
LINE3	2.9 - 5.0	G395M, G395H
LINE4	0.6 - 5.0	PRISM
REF	1.3 - 1.7	G140M, G140H

The pairing comes because the calibration unit lightpath doesn't pass through the filter wheel, so each lamp has its own filter identical to those in the filter wheel.

The list of `Spec2Pipeline` steps to be run for NIRSpec lamp exposures is shown in the table below and indicates which steps are applied to various spectroscopic modes. The instrument mode abbreviations used in the table are as follows:

- NIRSpec FS = Fixed Slit (also Brightobj)
- NIRSpec MOS = Multi-Object Spectroscopy
- NIRSpec IFU = Integral Field Unit

Pipeline Step	NRS_LAMP		NRS_AUTOWAVE	NRS_AUTOFLAT (MOS only)
	LINE	FLAT		
assign_wcs	ALL	ALL	ALL	ALL
msaflagopen	MOS, IFU	MOS, IFU	MOS, IFU	MOS
nsclean	NONE	NONE	NONE	NONE
imprint	NONE	IFU	NONE	NONE
background	NONE	NONE	NONE	NONE
extract_2d	MOS, FS	MOS, FS	MOS, FS	MOS
srctype	NONE	NONE	NONE	NONE
wavecorr	ALL	ALL	ALL	ALL
flat_field				NONE
• D-FLAT	ALL	ALL	ALL	
• S-FLAT	ALL	NONE	ALL	
• F-FLAT	NONE	NONE	NONE	
pathloss	NONE	NONE	NONE	NONE
barshadow	NONE	NONE	NONE	NONE
photom	NONE	NONE	NONE	NONE
resample_spec	MOS, FS	NONE	MOS, FS	NONE
cube_build	IFU	NONE	IFU	NONE
extract_1d	ALL	NONE	ALL	NONE

In the *resample_spec* and *cube_build* steps, the spectra are transformed to a space of (wavelength, offset along the slit) without applying a tangent plane projection.

Arguments

The `calwebb_spec2` pipeline has two optional arguments.

`--save_bsub` (boolean, default=False)
If set to `True`, the results of the background subtraction step will be saved to an intermediate file, using a product type of "_bsub" or "_bsubints", depending on whether the data are 2D (averaged over integrations) or 3D (per-integration results).

`--save_wfss_esec` (boolean, default=False)
If set to `True`, an intermediate image product is created for WFSS exposures that is in units of electrons/sec, instead of the normal DN/sec units that are used throughout the rest of processing. This product can be useful for doing off-line specialized processing of WFSS images. This product is created after the *background* and *flat-field* steps have been applied, but before the *extract_2d* step, so that it is the full WFSS image. The conversion to units of electrons/sec is accomplished by loading the GAIN reference file, computing the mean gain across all pixels (excluding reference pixels), and multiplying the WFSS image by the mean gain. The intermediate file will have a product type of "_esec". Only applies to WFSS exposures.

Inputs

2D or 3D countrate data

Data model
ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamodels.
or CubeModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMo

File suffix
_rate or _rateints

The input to the `Spec2Pipeline` pipeline is a countrate exposure, in the form of either "_rate" or "_rateints" data. A single input file can be processed or an ASN file listing multiple inputs can be used, in which case the processing steps will be applied to each input exposure, one at a time.

If "_rateints" products are used as input, for modes other than NIRSpec Fixed Slit, each step applies its algorithm to each integration in the exposure, where appropriate. For the NIRSpec Fixed Slit mode the `calwebb_spec2` pipeline will currently skip both the *resample_spec* step and the *extract_1d* step, because neither step supports multiple integration input products for this mode.

Note that the steps *background* and *imprint* can only be executed when the pipeline is given an ASN file as input, because they rely on multiple, associated exposures to perform their tasks. The ASN file must list not only the input science exposure(s), but must also list the exposures to be used as background or imprint.

Background subtraction for Wide-Field Slitless Spectroscopy (WFSS) exposures, on the other hand, is accomplished by scaling and subtracting a master background image contained in a CRDS reference file and hence does not require an ASN as input.

The input data model type IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jws
is only used for MIRI MRS and NIRSpec IFU exposures.

Outputs

2D or 3D background-subtracted data

Data model
ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamodels.
or CubeModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMo

File suffix
_bsub or _bsubints

This is an intermediate product that is only created if "–save_bsub" is set to `True` and will contain the data as output from the *background* step. If the input is a "_rate" product, this will be a "_bsub" product, while "_rateints" inputs will be saved as "_bsubints."

2D or 3D calibrated data

Data model
ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamodels.
CubeModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode
SlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.SlitModel.html#jwst.datamodels.SlitModel),
or MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamode

File suffix
_cal or _calints

The output is a fully calibrated, but unrectified, exposure, using the product type suffix "_cal" or "_calints", depending on the type of input, e.g. "jw80600012001_02101_00003_mirimage_cal.fits." This is the output of the *photom* step, or whichever step is performed last before applying either *resample_spec*, *cube_build*, or *extract_1d*.

The output data model type can be any of the 4 listed above and is completely dependent on the type of input data and the observing mode. For data sets that do **not** go through *extract_2d* processing, the output will be either a ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageModel), IFUImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamodels.IFUImageM or CubeModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeModel), matching the corresponding input data type.

Of the data types that do go through *extract_2d* processing, the output type will consist of either a single slit model or a multi-slit model:

- NIRSpec Bright-Object and NIRCam TSO Grism: SlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.S
- NIRSpec Fixed Slit and MOS, as well as WFSS: MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamodels.MultiSlitModel)

The multi-slit model is simply an array of multiple slit models, each one containing the data and relevant meta data for a particular extracted slit or source. A MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamodels.MultiSlitModel) product will contain multiple tuples of SCI, ERR, DQ, WAVELENGTH, etc. arrays; one for each of the extracted slits/sources.

2D resampled data

Data model
SlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.SlitModel.html#jwst.datamodels.SlitModel)
or MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamode

File suffix
_s2d

If the input is a 2D exposure type that gets resampled/rectified by the *resample_spec* step, the rectified 2D spectral product is saved as a "_s2d" file. This image is intended for use as a quick-look product only and is not used in subsequent processing. The 2D unresampled, calibrated ("_cal") products are passed along as input to subsequent Stage 3 processing.

If the input to the *resample_spec* step is a MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitM then the resampled output will be in the form of a MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.Mu which contains an array of individual models, one per slit. Otherwise the output will be a single SlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.SlitModel.html#jwst.datamodels.SlitModel).

3D resampled (IFU cube) data

Data model
`IFUCubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUCubeModel.html#jwst.datamodels.IF

File suffix
_s3d

If the data are NIRSpec IFU or MIRI MRS, the result of the *cube_build* step will be 3D IFU spectroscopic cube saved to a "_s3d" file. The IFU cube is built from the data contained in a single exposure and is intended for use as a quick-look product only. The 2D unresampled, calibrated ("_cal") products are passed along as input to subsequent Stage 3 processing.

1D extracted spectral data

Data model
`MultiSpecModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSpecModel.html#jwst.datamodels

File suffix
_x1d or _x1dints

All types of inputs result in a 1D extracted spectral data product, which is saved as a "_x1d" or "_x1dints" file, depending on the input type. Observing modes such as MIRI LRS fixed slit and MRS, NIRCam and NIRISS WFSS, and NIRSpec fixed slit, MOS, and IFU result in an "_x1d" product containing extracted spectral data for one or more slits/sources. TSO modes, such as MIRI LRS slitless, NIRCam TSO grism, NIRISS SOSS, and NIRSpec Bright Object, for which the data are 3D stacks of integrations, result in "_x1dints" products containing extracted spectral data for each integration with the exposure.

PARS-SPEC2PIPELINE Parameter Reference File

REFTYPE
PARS-SPEC2PIPELINE

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-spec2pipeline references based on the following keywords.

Instrument	Keywords
MIRI	TSOVISIT
NIRCAM	CROWDFLD, EXP_TYPE, TSOVISIT
NIRISS	CROWDFLD, EXP_TYPE, TSOVISIT
NIRSPEC	EXP_TYPE, LAMP, OPMODE, TSOVISIT

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

calwebb_image3: Stage 3 Imaging Processing

Class
jwst.pipeline.Image3Pipeline

Alias
calwebb_image3

Stage 3 processing for direct-imaging observations is intended for combining the calibrated data from multiple exposures (e.g. a dither or mosaic pattern) into a single rectified (distortion corrected) product. Before being combined, the exposures receive additional corrections for the purpose of astrometric alignment, background matching, and outlier rejection. The steps applied by the `calwebb_image3` pipeline are shown below. This pipeline is intended for non-TSO imaging only. TSO imaging data should be processed using the *calwebb_tso3* pipeline.

calwebb_image3
assign_mtwcs
tweakreg
skymatch
outlier_detection
resample
source_catalog

Arguments

The `calwebb_image3` pipeline does not have any optional arguments.

Inputs

2D calibrated images

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_cal

The inputs to the `calwebb_image3` pipeline are one or more *calwebb_image2* calibrated ("_cal") image products. In order to process and combine multiple images, an ASN file must be used as input, listing the exposures to be processed. It is also possible use a single "_cal" file as input to `calwebb_image3`, in which case only the *resample* and *source_catalog* steps will be applied.

Outputs

CR-flagged exposures

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_crf

If the *outlier_detection* step is applied, a new version of each input calibrated exposure is created, in which the DQ array has been updated to flag pixels detected as outliers. These files use the "_crf" (CR-Flagged) product type suffix and also includes the association candidate ID as a new field in the original product root name, e.g. "jw96090001001_03101_00001_nrca2_o001_crf.fits."

Resampled and combined 2D image

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_i2d

A resampled 2D image product of type "_i2d" is created containing the combined, rectified association of exposures, which is the direct output of the *resample* step.

Source catalog

Data model
N/A

File suffix
_cat

The source catalog produced by the *source_catalog* step from the "_i2d" product is saved as an ASCII file in `ecsv` format, with a product type of "_cat."

Segmentation map

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_segm

A 2D image segmentation map produced by the *source_catalog* step from the "_i2d" product, saved as a FITS file with a single image extension and a product type suffix of "_segm."

calwebb_spec3: Stage 3 Spectroscopic Processing

Class
jwst.pipeline.Spec3Pipeline

Alias
calwebb_spec3

Stage 3 processing for spectroscopic observations is intended for combining the calibrated data from multiple exposures (e.g. a dither/nod pattern) into a single combined 2D or 3D spectral product and a combined 1D spectrum. Before being combined, the exposures may receive additional corrections for the purpose of background matching and subtraction, as well as outlier rejection. The steps applied by the `calwebb_spec3` pipeline are shown below. This pipeline is intended for non-TSO spectra only. TSO spectral data should be processed using the *calwebb_tso3* pipeline.

Instrument/Mode	NIRSpec			MIRI		NIRISS		NIRCam
Step	FS	MOS	IFU	FS	MRS	SOSS	WFSS	WFSS
assign_mtwcs[1]	✓	✓	✓	✓	✓	✓	✓	✓
master_background[2]	✓		✓	✓	✓			
exp_to_source	✓	✓					✓	✓
mrs_imatch					✓			
outlier_detection	✓	✓	✓	✓	✓			
resample_spec	✓	✓		✓				
cube_build			✓		✓			
extract_1d	✓	✓	✓	✓	✓	✓	✓	✓
spectral_leak					✓			
combine_1d						✓	✓	✓

[1]The *assign_mtwcs* step is only applied to observations of a moving target (TARGTYPE='moving').

[2]The master background subtraction step is applied to NIRSpec MOS exposures in the *calwebb_spec2* pipeline.

Notice that NIRCam and NIRISS WFSS, as well as NIRISS SOSS data, receive only minimal processing by `calwebb_spec3`. WFSS 2D input data are reorganized into source-based products by the *exp_to_source* step (see

below), have 1D extracted spectra produced for each source, and then the 1D spectra for each source are combined into a final 1D spectrum. NIRISS SOSS inputs do not go through the *exp_to_source* step, because they contain data for a single source. Hence the only processing that they receive is to extract a 1D spectrum from each input and then combine those spectra into a final 1D spectrum. This type of processing is intended only for NIRISS SOSS exposures that are not obtained in TSO mode. TSO mode NIRISS SOSS exposures should be processed with the *calwebb_tso3* pipeline.

Arguments

The `calwebb_spec3` pipeline does not have any optional arguments.

Inputs

2D calibrated data

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM
`IFUImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.datamodels.
`SlitModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.SlitModel.html#jwst.datamodels.SlitModel),
or `MultiSlitModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamode

File suffix
_cal

The inputs to `calwebb_spec3` should be in the form of an ASN file that lists the multiple exposures to be processed into combined output products. The individual exposures should be calibrated the ("_cal") products from *calwebb_spec2* processing.

The member list for each product in the ASN file can also contain exposures of dedicated background targets, which are intended for use in the *master_background* step. These input exposures must be the "x1d" products (extracted 1-D spectra) of the background target(s) and are usually the "x1d" files produced by the *calwebb_spec2* pipeline. They must be listed in the ASN file with "exptype" values of "background" in order to be correctly identified as background exposures. See the *master_background* for more details.

Outputs

Source-based calibrated data

Data model
`MultiExposureModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiExposureModel.html#jwst.d

File suffix
_cal

For NIRSpec fixed-slit, NIRSpec MOS, and NIRCam and NIRISS WFSS, which have a defined set of slits or sources, the data from the input calibrated exposures is reorganized by the *exp_to_source* step so that all of the instances of data for a particular source/slit are contained in a single product. These are referred to as "source-based" products, as opposed to the input exposure-based products. The source-based collections of data are saved in intermediate files, one per source/slit. The root names of the source-based files contain the source ID as an identifier and use the same "_cal" suffix as the input calibrated exposure files. An example source-based file name is "jw00042-o001_s0002_niriss_gr150r_f150w_cal.fits", where "s0002" is the source id.

The reorganized sets of data are sent to subsequent steps to process and combine all the data for one source at a time.

CR-flagged exposures

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_crf

If the *outlier_detection* step is applied, a new version of each input calibrated exposure is created, in which the DQ array has been updated to flag pixels detected as outliers. These files use the "_crf" (CR-Flagged) product type suffix and also includes the association candidate ID as a new field in the original product root name, e.g. "jw96090001001_03101_00001_nrs2_o001_crf.fits."

2D resampled and combined spectral data

Data model
`SlitModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.SlitModel.html#jwst.datamodels.SlitModel)

File suffix
_s2d

When processing non-IFU modes, a resampled/rectified 2D product of type "_s2d" is created containing the rectified and combined data for a given slit/source, which is the output of the *resample_spec* step.

3D resampled and combined spectral data

Data model
`IFUCubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUCubeModel.html#jwst.datamodels.IF

File suffix
_s3d

When processing IFU exposures, a resampled and combined 3D IFU cube product created by the *cube_build* step is saved as an "_s3d" file.

1D extracted spectral data

Data model
`MultiSpecModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSpecModel.html#jwst.datamodels

File suffix
_x1d

All types of inputs result in a 1D extracted spectral data product, which is saved as a "_x1d" file, and is normally the result of performing the *extract_1d* step on the combined "_s2d" or "_s3d" product.

For NIRCam and NIRISS WFSS, as well as NIRISS SOSS data, the *extract_1d* is performed on the individual unresampled 2D cutout images, resulting in multiple 1-D spectra per source in a "_x1d" product. Those spectra are combined using the subsequent *combine_1d* step (see below).

1D combined spectral data

Data model
`CombinedSpecModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CombinedSpecModel.html#jwst.da

File suffix
_c1d

For NIRCam and NIRISS WFSS, as well as NIRISS SOSS data, the *combine_1d* combines the multiple 1-D spectra for a given source into a final spectrum, which is saved as a "_c1d" product.

calwebb_ami3: Stage 3 Aperture Masking Interferometry (AMI) Processing

Class
jwst.pipeline.Ami3Pipeline

Alias
calwebb_ami3

The stage 3 AMI pipeline is applied to associations of calibrated NIRISS AMI exposures. It computes fringe parameters for individual exposures, averages the fringe results from multiple exposures, and, optionally, corrects science target fringe parameters using the fringe results from reference PSF targets. The steps applied by the `calwebb_ami3` pipeline are shown below.

calwebb_ami3
ami_analyze
ami_average
ami_normalize

When given an association file as input, which lists multiple science target and reference PSF exposures, the pipeline will:

1. apply the *ami_analyze* step to each input exposure independently, computing fringe parameters for each
2. apply the *ami_average* step to compute the average of the *ami_analyze* results for all of the science target exposures, and the average for all of the reference PSF results (if present)
3. apply the *ami_normalize* step to correct the average science target results using the average reference PSF results (if present)

If no reference PSF target exposures are present in the input ASN file, the `ami_normalize` step is skipped.

Arguments

The `calwebb_ami3` pipeline has one optional argument:

```
--save_averages  boolean  default=False
```

If set to `True`, the results of the *ami_average* step will be saved to a file. If not, the results of the *ami_average* step are passed along in memory to the *ami_normalize* step.

Inputs

2D calibrated images

Data model
ImageModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_cal

The inputs to `calwebb_ami3` need to be in the form of an ASN file that lists multiple science target exposures, and optionally reference PSF exposures as well. The individual exposures must be in the form of calibrated ("_cal") products from *calwebb_image2* processing.

An example ASN file containing 2 science target and 2 reference PSF target exposures is shown below. Only 1 product is defined, corresponding to the science target, with members consisting of exposures for both the science target and the reference PSF target, as indicated by the "exptype" values for each.

```
{"asn_type": "ami3",
 "asn_rule": "discover_Asn_AMI",
 "program": "10005",
 "asn_id": "a3001",
 "target": "t001",
 "asn_pool": "jw10005_001_01_pool",
 "products": [
     {"name": "jw10005-a3001_t001_niriss_f277w-nrm",
      "members": [
          {"expname": "jw10005007001_02102_00001_nis_cal.fits",
           "exptype": "psf"
          },
          {"expname": "jw10005027001_02102_00001_nis_cal.fits",
           "exptype": "psf"
          },
          {"expname": "jw10005004001_02102_00001_nis_cal.fits",
           "exptype": "science"
          },
          {"expname": "jw10005001001_02102_00001_nis_cal.fits",
           "exptype": "science"
          }
      ]
     }
 ]
}
```

Outputs

Fringe parameter tables

Data model
AmiLgModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_ami

For every input exposure, the fringe parameters and closure phases calculated by the *ami_analyze* step are saved to an "_ami" product file, which is a FITS table containing the fringe parameters and closure phases. Product names use the input "_cal" exposure-based file name, with the association candidate ID included and the product type changed to "_ami", e.g. "jw93210001001_03101_00001_nis_a0003_ami.fits."

Averaged fringe parameters table

Data model
`AmiLgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_amiavg or _psf-amiavg

If multiple target or reference PSF exposures are used as input and the "–save_averages" parameter is set to `True`, the *ami_average* step will save averaged results for the target in an "_amiavg" product and for the reference PSF in a "_psf-amiavg" product. The file name root will use the source-based output product name given in the ASN file. These files are the same FITS table format as the "_ami" products.

Normalized fringe parameters table

Data model
`AmiLgModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.AmiLgModel.html#jwst.datamodels.AmiLg

File suffix
_aminorm

If reference PSF exposures are included in the input ASN, the averaged AMI results for the target will be normalized by the averaged AMI results for the reference PSF, via the *ami_normalize* step, and will be saved to an "_aminorm" product file. This file has the same FITS table format as the "_ami" products. The file name root uses the source-based output product name given in the ASN file, e.g. "jw93210-a0003_t001_niriss_f480m-nrm_aminorm.fits."

calwebb_coron3: Stage 3 Coronagraphic Processing

Class
jwst.pipeline.Coron3Pipeline

Alias
calwebb_coron3

The stage 3 coronagraphic pipeline is to be applied to associations of calibrated NIRCam coronagraphic and MIRI Lyot and 4QPM exposures, and is used to produce PSF-subtracted, resampled, combined images of the source object.

The steps applied by the `calwebb_coron3` pipeline are shown in the table below.

calwebb_coron3
outlier_detection
stack_refs
align_refs
klip
resample

The high-level processing provided by these steps is:

1) CR-flag all PSF and science target exposures

2) Accumulate all reference PSF images into a single product

3) Align every PSF image to every science target image

4) Compute an optimal PSF fit and subtract it from every science target image

5) Combine the PSF-subtracted and CR-flagged images into a single resampled image

Currently the individual steps shown above can only be run in a convenient way by running the `calwebb_coron3` pipeline on an association (ASN) file that lists the various science target and reference PSF exposures to be processed.

Arguments

The `calwebb_coron3` pipeline does not have any optional arguments.

Inputs

3D calibrated images

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_calints

The input to `calwebb_coron3` must be in the form of an ASN file that lists one or more exposures of a science target and one or more reference PSF targets. The individual target and reference PSF exposures should be in the form of 3D calibrated ("_calints") products from *calwebb_image2* processing. Each pipeline step will loop over the 3D stack of per-integration images contained in each exposure.

An example ASN file containing 2 science target and 1 reference PSF target exposures is shown below. Only 1 product is defined, corresponding to the science target, with members consisting of exposures of both the science target and the reference PSF target, as indicated by the "exptype" values for each:

```
{"asn_type": "coron3",
 "asn_rule": "candidate_Asn_Coron",
 "program": "10005",
 "asn_id": "c1001",
 "target": "t001",
 "asn_pool": "jw10005_20181020T033546_pool",
 "products": [
     {"name": "jw10005-c1001_t001_nircam_f430m-maskrnd-sub320a430r",
      "members": [
          {"expname": "jw10005009001_02102_00001_nrcalong_calints.fits",
           "exptype": "psf"
          },
          {"expname": "jw10005006001_02102_00001_nrcalong_calints.fits",
           "exptype": "science"
          },
          {"expname": "jw10005003001_02102_00001_nrcalong_calints.fits",
           "exptype": "science"
          }
      ]
     }
```

(continues on next page)

(continued from previous page)

```
 ]
}
```

Outputs

CR-flagged images

Data model
CubeModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_crfints

If the *outlier_detection* step is applied, a new version of each exposure is created, in which the DQ array is updated to flag pixels detected as outliers. These files use the "_crfints" (CR-Flagged per integration) product type suffix and include the association candidate ID, e.g. "jw8607342001_02102_00001_nrcb3_a3001_crfints.fits."

3D stacked PSF images

Data model
CubeModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_psfstack

The data from each input PSF reference exposure are concatenated into a single combined 3D stack by the *stack_refs* step, for use by subsequent steps. The stacked PSF data get written to disk in the form of a "_psfstack" product. The output file name is source-based, using the product name specified in the ASN file, e.g. "jw86073-a3001_t001_nircam_f140m-maskbar_psfstack.fits."

4D aligned PSF images

Data model
QuadModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.QuadModel.html#jwst.datamodels.QuadMod

File suffix
_psfalign

For each science target exposure, all of the reference PSF images in the "_psfstack" product are aligned to each science target integration and saved to a 4D "_psfalign" product by the *align_refs* step. The output file name is exposure-based, with the addition of the associated candidate ID, e.g. "jw8607342001_02102_00001_nrcb3_a3001_psfalign.fits."

3D PSF-subtracted images

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_psfsub

For each science target exposure, the *klip* step applies PSF fitting and subtraction for each integration, resulting in a 3D stack of PSF-subtracted images. The data for each science target exposure are saved to a "_psfsub" product, using exposure-based file names, e.g. "jw8607342001_02102_00001_nrcb3_a3001_psfsub.fits."

2D resampled image

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_i2d

The *resample* step is applied to the CR-flagged products to create a single resampled and combined product for the science target. The file name is source-based, using the product name specified in the ASN file, e.g. "jw86073-a3001_t001_nircam_f140m-maskbar_i2d.fits."

calwebb_tso3: Stage 3 Time-Series Observation(TSO) Processing

Class
jwst.pipeline.Tso3Pipeline

Alias
calwebb_tso3

The stage 3 TSO pipeline is to be applied to associations of calibrated TSO exposures (e.g. NIRCam TS imaging, NIRCam TS grism, NIRISS SOSS, NIRSpec BrightObj, MIRI LRS Slitless) and is used to produce calibrated time-series photometry or spectra of the source object.

The steps applied by the `calwebb_tso3` pipeline for Imaging and Spectroscopic TSO exposures are shown below:

calwebb_tso3	Imaging	Spectroscopy
outlier_detection	✓	✓
tso_photometry	✓	
extract_1d		✓
white_light		✓

The logic that decides whether to apply the imaging or spectroscopy steps is based on the EXP_TYPE and TSOVISIT keyword values of the input data. Imaging steps are applied if either of the following is true:

- EXP_TYPE = 'NRC_TSIMAGE'
- EXP_TYPE = 'MIR_IMAGE' and TSOVISIT = True

The spectroscopy steps will be applied in all other cases.

Inputs

3D calibrated images

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_calints

The input to `calwebb_tso3` is in the form of an ASN file that lists multiple exposures or exposure segments of a science target. The individual inputs should be in the form of 3D calibrated ("_calints") products from either *calwebb_image2* or *calwebb_spec2* processing. These products contain 3D stacks of per-integration images. Each pipeline step will loop over all of the integrations in each input.

Many TSO exposures may contain a sufficiently large number of integrations (NINTS) so as to make their individual exposure products too large (in terms of file size on disk) to be able to handle conveniently. In these cases, the uncalibrated raw data for a given exposure are split into multiple "segmented" products, each of which is identified with a segment number (see *segmented products*). The `calwebb_tso3` input ASN file includes all "_calints" exposure segments. The *outlier_detection* step will process a single segment at a time, creating one output "_crfints" product per segment. The remaining `calwebb_tso3` steps, will process each segment and concatenate the results into a single output product, containing the results for all exposures and segments listed in the ASN.

Outputs

3D CR-flagged images

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_crfints

If the *outlier_detection* step is applied, a new version of each input calibrated product is created, which contains a DQ array that has been updated to flag pixels detected as outliers. This updated product is known as a CR-flagged product and is saved as a "_crfints" product type.

Imaging photometry

Data model
N/A

File suffix
_phot

For imaging TS observations, the *tso_photometry* step produces a source catalog containing photometry results from all of the "_crfints" products, organized as a function of integration time stamps. This file is saved in ASCII "ecsv" format, with a product type of "_phot." The file naming is source-based, using the output product name specified in the ASN file, e.g. "jw93065-a3001_t1_nircam_f150w-wlp8_phot.ecsv."

1D extracted spectral data

Data model
`MultiSpecModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSpecModel.html#jwst.datamodels

File suffix
_x1dints

For spectroscopic TS observations, the *extract_1d* step is applied to all "_crfints" products, to create a single "_x1dints" product that contains 1D extracted spectral data for all integrations contained in the input exposures. The file name is source-based, using the output product name specified in the ASN file, e.g. "jw87600-a3001_t001_niriss_clear-gr700xd_x1dints.fits."

Spectroscopic white-light photometry

Data model
N/A

File suffix
_whtlt

For spectroscopic TS observations, the *white_light* step is applied to all of the 1D extracted spectral data in the "_x1dints" product, to produce an ASCII catalog in `ecsv` format containing the wavelength-integrated white-light photometry of the source. The catalog lists the integrated white-light flux as a function of time, based on the integration time stamps. The file name is source-based, using the output product name specified in the ASN file, e.g. "jw87600-a3001_t001_niriss_clear-gr700xd_whtlt.ecsv."

calwebb_dark: Dark Processing

Class
jwst.pipeline.DarkPipeline

Alias
calwebb_dark

The `DarkPipeline` applies basic detector-level corrections to all dark exposures. It is identical to the *calwebb_detector1* pipeline, except that it stops processing immediately before the *dark_current* step. The list of steps is shown below. As with the *calwebb_detector1* pipeline, the order of steps is a bit different for MIRI exposures.

Near-IR	MIRI
group_scale	*group_scale*
dq_init	*dq_init*
	emicorr
saturation	*saturation*
ipc[1]	*ipc*
superbias	*firstframe*
refpix	*lastframe*
linearity	*reset*
	linearity
	rscd

[1] By default, the parameter reference `pars-darkpipeline` retrieved from CRDS will skip the *ipc* step.

Arguments

The `calwebb_dark` pipeline has no optional arguments.

Inputs

4D raw data

Data model
`RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamodels.RampMo

File suffix
_uncal

The input to `DarkPipeline` is a single raw dark exposure, which contains the original raw data from all of the detector readouts in the exposure (ncols x nrows x ngroups x nintegrations).

Outputs

4D corrected ramp

Data model
`RampModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RampModel.html#jwst.datamodels.RampMo

File suffix
_dark

Result of applying all pipeline steps listed above. Will have the same data dimensions as the input raw 4D data (ncols x nints x ngroups x nints).

PARS-DARKPIPELINE Reference File

REFTYPE
PARS-DARKPIPELINE

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-whitelightstep references based on the following keywords.

Instrument	Keywords
FGS	INSTRUME
MIRI	INSTRUME
NIRCAM	INSTRUME
NIRISS	INSTRUME
NIRSPEC	INSTRUME

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

calwebb_guider: Guide Star Processing

Class
jwst.pipeline.GuiderPipeline

Alias
calwebb_guider

The guider pipeline is only for use with data resulting from the FGS guiding functions: Identification (ID), Acquisition (ACQ1 and ACQ2), Track, and Fine Guide. The pipeline applies three detector-level correction and calibration steps to uncalibrated guider data, as listed in the table below.

calwebb_guider
dq_init
guider_cds
flat_field

Arguments

The `calwebb_guider` pipeline does not have any optional arguments.

Inputs

4D raw data

Data model
GuiderRawModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.GuiderRawModel.html#jwst.datamode

File suffix
_uncal

The input to `calwebb_guider` is a single raw guide-mode data file, which contains the original raw data from all of the detector readouts performed during the guider mode episode. The organization of the 4D data array is analogous to that of 4D raw science data, having dimensions of ncols x nrows x ngroups x nintegrations.

Outputs

3D calibrated data

Data model
GuiderCalModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.GuiderCalModel.html#jwst.datamodels

File suffix
_cal

The output is a 3D (ncols x nrows x nints) countrate product that has been flat-fielded and has bad pixels flagged. For ID mode data, there is only 1 countrate image produced by the *guider_cds* step, therefore the length of the 3rd array axis is 1. For all other modes, there will be a stack of multiple countrate images, one per integration. See the *guider_cds* step information for details on how the countrate images are produced for each mode.

calwebb_wfs-image2: Stage 2 WFS&C Processing

Deprecated post-1.1.0
The operation of the pipeline is no longer dependent on built-in configuration files. How *jwst.pipeline.Image2Pipeline* processes WFS&C data is determined by the CRDS reftype `pars-image2pipeline`. The version of `calwebb_wfs-image2.cfg` delivered with the software is devoid of any configuration and will be removed in a future version.

Config
calwebb_wfs-image2.cfg

Class
jwst.pipeline.Image2Pipeline

Stage 2 processing of Wavefront Sensing and Control (WFS&C) images duplicates the processing applied to regular science imaging, with the exception of image resampling. The `calwebb_wfs-image2.cfg` configuration utilizes the regular `Image2Pipeline` module, with the *resample* step set to be skipped, because the analysis of WFS&C data must be done in the original unrectified image space. The list of steps is shown in the table below.

calwebb_image2	calwebb_wfs-image2
background	✓
assign_wcs	✓
flat_field	✓
photom	✓
resample	

Arguments

The `calwebb_wfs-image2` pipeline does not have any optional arguments.

Inputs

2D countrate data

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_rate

The input to `Image2Pipeline` is a countrate exposure, in the form of "_rate" data. A single input file can be processed or an ASN file listing multiple inputs can be used, in which case the processing steps will be applied to each input exposure, one at a time.

Outputs

2D calibrated data

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_cal

The output is a fully calibrated, but unrectified, exposure, using the product type suffix "_cal."

calwebb_wfs-image3: Stage 3 WFS&C Processing

Class
jwst.wfs_combine.WfsCombineStep

Alias
calwebb_wfs-image3

Stage 3 processing of Wavefront Sensing and Control (WFS&C) images is only performed for dithered pairs of WFS&C exposures. The processing applied is not truly a "pipeline", but consists only of the single *wfs_combine* step. The `calwebb_wfs-image3` alias exists only for consistency and compatibility with stage 3 processing of other observing modes. The same result could be obtained by just running the *wfs_combine* step directly.

Arguments

The `calwebb_wfs-image3` pipeline has one optional argument:

```
--do_refine  boolean  default=False
```

If set to `True`, offsets between the dithered images computed from the WCS will be refined empirically using a cross-correlation technique. See *wfs_combine* for details.

Inputs

2D calibrated images

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_cal

The input to `calwebb_wfs-image3` is a pair of calibrated ("_cal") exposures, specified via an ASN file.

Outputs

2D combined image

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_wfscmb

The output is a combined image, using the product type suffix "_wfscmb." See *wfs_combine* for details on how this combined image is produced.

jwst.pipeline Package

Classes

Ami3Pipeline(*args, **kwargs)	Ami3Pipeline: Apply all level-3 calibration steps to an association of level-2b AMI exposures.
Coron3Pipeline(*args, **kwargs)	Class for defining Coron3Pipeline.
DarkPipeline(*args, **kwargs)	DarkPipeline: Apply detector-level calibration steps to raw JWST dark ramp to produce a corrected 4-D ramp product.
Detector1Pipeline(*args, **kwargs)	Detector1Pipeline: Apply all calibration steps to raw JWST ramps to produce a 2-D slope product.
GuiderPipeline(*args, **kwargs)	GuiderPipeline: For FGS observations, apply all calibration steps to raw JWST ramps to produce a 3-D slope product.
Image2Pipeline(*args, **kwargs)	Image2Pipeline: Processes JWST imaging-mode slope data from Level-2a to Level-2b.
Image3Pipeline(*args, **kwargs)	Image3Pipeline: Applies level 3 processing to imaging-mode data from
Spec2Pipeline(*args, **kwargs)	Spec2Pipeline: Processes JWST spectroscopic exposures from Level 2a to 2b.
Spec3Pipeline(*args, **kwargs)	Spec3Pipeline: Processes JWST spectroscopic exposures from Level 2b to 3.
Tso3Pipeline(*args, **kwargs)	TSO3Pipeline: Applies level 3 processing to TSO-mode data from

Ami3Pipeline

class jwst.pipeline.**Ami3Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

Ami3Pipeline: Apply all level-3 calibration steps to an association of level-2b AMI exposures. Included steps are: ami_analyze (fringe detection) ami_average (average results of fringe detection) ami_normalize (normalize results by reference target)

See Step.__init__ for the parameters.

Attributes Summary

class_alias
spec
step_defs

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'calwebb_ami3'

spec

```
save_averages = boolean(default=True)
```

step_defs = {'ami_analyze': <class 'jwst.ami.ami_analyze_step.AmiAnalyzeStep'>, 'ami_average': <class 'jwst.ami.ami_average_step.AmiAverageStep'>, 'ami_normalize': <class 'jwst.ami.ami_normalize_step.AmiNormalizeStep'>}

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Coron3Pipeline

class jwst.pipeline.**Coron3Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

Class for defining Coron3Pipeline.

Coron3Pipeline: Apply all level-3 calibration steps to a coronagraphic association of exposures. Included steps are:

1. stack_refs (assemble reference PSF inputs)
2. align_refs (align reference PSFs to target images)
3. klip (PSF subtraction using the KLIP algorithm)
4. outlier_detection (flag outliers)
5. resample (image combination and resampling)

See Step.__init__ for the parameters.

Attributes Summary

class_alias
prefetch_references
spec
step_defs

Methods Summary

process(user_input)	Primary method for performing pipeline.

Attributes Documentation

class_alias = 'calwebb_coron3'

prefetch_references = False

spec

```
suffix = string(default='i2d')
```

step_defs = {'align_refs': <class 'jwst.coron.align_refs_step.AlignRefsStep'>, 'klip': <class 'jwst.coron.klip_step.KlipStep'>, 'outlier_detection': <class 'jwst.outlier_detection.outlier_detection_step.OutlierDetectionStep'>, 'resample': <class 'jwst.resample.resample_step.ResampleStep'>, 'stack_refs': <class 'jwst.coron.stack_refs_step.StackRefsStep'>}

Methods Documentation

process(*user_input*)

Primary method for performing pipeline.

Parameters

user_input (*str* (https://docs.python.org/3/library/stdtypes.html#str), *Level3 Association*, *or* *JwstDataModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataM – The exposure or association of exposures to process

DarkPipeline

class jwst.pipeline.**DarkPipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

DarkPipeline: Apply detector-level calibration steps to raw JWST dark ramp to produce a corrected 4-D ramp product. Included steps are: group_scale, dq_init, saturation, ipc, superbias, refpix, rscd, lastframe, and linearity.

See Step.__init__ for the parameters.

Attributes Summary

class_alias	
step_defs	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'calwebb_dark'

step_defs = {'dq_init': <class 'jwst.dq_init.dq_init_step.DQInitStep'>, 'emicorr': <class 'jwst.emicorr.emicorr_step.EmiCorrStep'>, 'firstframe': <class 'jwst.firstframe.firstframe_step.FirstFrameStep'>, 'group_scale': <class 'jwst.group_scale.group_scale_step.GroupScaleStep'>, 'ipc': <class 'jwst.ipc.ipc_step.IPCStep'>, 'lastframe': <class 'jwst.lastframe.lastframe_step.LastFrameStep'>, 'linearity': <class 'jwst.linearity.linearity_step.LinearityStep'>, 'refpix': <class 'jwst.refpix.refpix_step.RefPixStep'>, 'reset': <class 'jwst.reset.reset_step.ResetStep'>, 'rscd': <class 'jwst.rscd.rscd_step.RscdStep'>, 'saturation': <class 'jwst.saturation.saturation_step.SaturationStep'>, 'superbias': <class 'jwst.superbias.superbias_step.SuperBiasStep'>}

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Detector1Pipeline

class jwst.pipeline.**Detector1Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

Detector1Pipeline: Apply all calibration steps to raw JWST ramps to produce a 2-D slope product. Included steps are: group_scale, dq_init, saturation, ipc, superbias, refpix, rscd, lastframe, linearity, dark_current, persistence, jump detection, ramp_fit, and gain_scale.

See `Step.__init__` for the parameters.

Attributes Summary

class_alias	
spec	
step_defs	

Methods Summary

process(input)	This is where real work happens.
setup_output(input)	

Attributes Documentation

class_alias = 'calwebb_detector1'

spec

```
save_calibrated_ramp = boolean(default=False)
```

step_defs = {'charge_migration': <class 'jwst.charge_migration.charge_migration_step.ChargeMigrationStep'>, 'dark_current': <class 'jwst.dark_current.dark_current_step.DarkCurrentStep'>, 'dq_init': <class 'jwst.dq_init.dq_init_step.DQInitStep'>, 'emicorr': <class 'jwst.emicorr.emicorr_step.EmiCorrStep'>, 'firstframe': <class 'jwst.firstframe.firstframe_step.FirstFrameStep'>, 'gain_scale': <class 'jwst.gain_scale.gain_scale_step.GainScaleStep'>, 'group_scale': <class 'jwst.group_scale.group_scale_step.GroupScaleStep'>, 'ipc': <class 'jwst.ipc.ipc_step.IPCStep'>, 'jump': <class 'jwst.jump.jump_step.JumpStep'>, 'lastframe': <class 'jwst.lastframe.lastframe_step.LastFrameStep'>, 'linearity': <class 'jwst.linearity.linearity_step.LinearityStep'>, 'persistence': <class 'jwst.persistence.persistence_step.PersistenceStep'>, 'ramp_fit': <class 'jwst.ramp_fitting.ramp_fit_step.RampFitStep'>, 'refpix': <class 'jwst.refpix.refpix_step.RefPixStep'>, 'reset': <class 'jwst.reset.reset_step.ResetStep'>, 'rscd': <class 'jwst.rscd.rscd_step.RscdStep'>, 'saturation': <class 'jwst.saturation.saturation_step.SaturationStep'>, 'superbias': <class 'jwst.superbias.superbias_step.SuperBiasStep'>}

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

setup_output(*input*)

GuiderPipeline

class jwst.pipeline.**GuiderPipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

GuiderPipeline: For FGS observations, apply all calibration steps to raw JWST ramps to produce a 3-D slope product. Included steps are: dq_init, guider_cds, and flat_field.

See Step.__init__ for the parameters.

Attributes Summary

class_alias	
step_defs	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'calwebb_guider'

step_defs = {'dq_init': <class 'jwst.dq_init.dq_init_step.DQInitStep'>, 'flat_field': <class 'jwst.flatfield.flat_field_step.FlatFieldStep'>, 'guider_cds': <class 'jwst.guider_cds.guider_cds_step.GuiderCdsStep'>}

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Image2Pipeline

class jwst.pipeline.**Image2Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

Image2Pipeline: Processes JWST imaging-mode slope data from Level-2a to Level-2b.

Included steps are: background_subtraction, assign_wcs, flat_field, photom and resample.

See Step.__init__ for the parameters.

Attributes Summary

class_alias
image_exptypes
spec
step_defs

Methods Summary

process(input)	This is where real work happens.
process_exposure_product(exp_product[, ...])	Process an exposure found in the association product

Attributes Documentation

class_alias = 'calwebb_image2'

image_exptypes = ['MIR_IMAGE', 'NRC_IMAGE', 'NIS_IMAGE', 'FGS_IMAGE']

spec

```
save_bsub = boolean(default=False) # Save background-subtracted science
```

step_defs = {'assign_wcs': <class 'jwst.assign_wcs.assign_wcs_step.AssignWcsStep'>, 'bkg_subtract': <class 'jwst.background.background_step.BackgroundStep'>, 'flat_field': <class 'jwst.flatfield.flat_field_step.FlatFieldStep'>, 'photom': <class 'jwst.photom.photom_step.PhotomStep'>, 'resample': <class 'jwst.resample.resample_step.ResampleStep'>}

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

process_exposure_product(*exp_product*, *pool_name=' '*, *asn_file=' '*)

Process an exposure found in the association product

Parameters

- **exp_product** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – A Level2b association product.
- **pool_name** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The pool file name. Used for recording purposes only.
- **asn_file** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The name of the association file. Used for recording purposes only.

Image3Pipeline

class jwst.pipeline.**Image3Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

Image3Pipeline: Applies level 3 processing to imaging-mode data from any JWST instrument.

Included steps are:
assign_mtwcs tweakreg skymatch outlier_detection resample source_catalog

See Step.__init__ for the parameters.

Attributes Summary

class_alias	
spec	
step_defs	

Methods Summary

process(input_data)	Run the Image3Pipeline

Attributes Documentation

class_alias = 'calwebb_image3'

spec

step_defs = {'assign_mtwcs': <class 'jwst.assign_mtwcs.assign_mtwcs_step.AssignMTWcsStep'>, 'outlier_detection': <class 'jwst.outlier_detection.outlier_detection_step.OutlierDetectionStep'>, 'resample': <class 'jwst.resample.resample_step.ResampleStep'>, 'skymatch': <class 'jwst.skymatch.skymatch_step.SkyMatchStep'>, 'source_catalog': <class 'jwst.source_catalog.source_catalog_step.SourceCatalogStep'>, 'tweakreg': <class 'jwst.tweakreg.tweakreg_step.TweakRegStep'>}

Methods Documentation

process(*input_data*)

Run the Image3Pipeline

Parameters

input_data (*Level3 Association, or ModelContainer*) – The exposures to process

Spec2Pipeline

class jwst.pipeline.**Spec2Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

Spec2Pipeline: Processes JWST spectroscopic exposures from Level 2a to 2b. Accepts a single exposure or an association as input.

Included steps are: assign_wcs, NIRSpec MSA bad shutter flagging, nsclean, background subtraction, NIRSpec MSA imprint subtraction, 2-D subwindow extraction, flat field, source type decision, straylight, fringe, residual_fringe, pathloss, barshadow, photom, resample_spec, cube_build, and extract_1d.

See Step.__init__ for the parameters.

Attributes Summary

class_alias
spec
step_defs

Methods Summary

process(data)	Entrypoint for this pipeline
process_exposure_product(exp_product[, ...])	Process an exposure found in the association product

Attributes Documentation

class_alias = 'calwebb_spec2'

spec

```
save_bsub = boolean(default=False)        # Save background-subtracted science
fail_on_exception = boolean(default=True) # Fail if any product fails.
save_wfss_esec = boolean(default=False)   # Save WFSS e-/sec image
```

step_defs = {'assign_wcs': <class 'jwst.assign_wcs.assign_wcs_step.AssignWcsStep'>, 'barshadow': <class 'jwst.barshadow.barshadow_step.BarShadowStep'>, 'bkg_subtract': <class 'jwst.background.background_step.BackgroundStep'>, 'cube_build': <class 'jwst.cube_build.cube_build_step.CubeBuildStep'>, 'extract_1d': <class 'jwst.extract_1d.extract_1d_step.Extract1dStep'>, 'extract_2d': <class 'jwst.extract_2d.extract_2d_step.Extract2dStep'>, 'flat_field': <class 'jwst.flatfield.flat_field_step.FlatFieldStep'>, 'fringe': <class 'jwst.fringe.fringe_step.FringeStep'>, 'imprint_subtract': <class 'jwst.imprint.imprint_step.ImprintStep'>, 'master_background_mos': <class 'jwst.master_background.master_background_mos_step.MasterBackgroundMosStep'>, 'msa_flagging': <class 'jwst.msaflagopen.msaflagopen_step.MSAFlagOpenStep'>, 'nsclean': <class 'jwst.nsclean.nsclean_step.NSCleanStep'>, 'pathloss': <class 'jwst.pathloss.pathloss_step.PathLossStep'>, 'photom': <class 'jwst.photom.photom_step.PhotomStep'>, 'pixel_replace': <class 'jwst.pixel_replace.pixel_replace_step.PixelReplaceStep'>, 'resample_spec': <class 'jwst.resample.resample_spec_step.ResampleSpecStep'>, 'residual_fringe': <class 'jwst.residual_fringe.residual_fringe_step.ResidualFringeStep'>, 'srctype': <class 'jwst.srctype.srctype_step.SourceTypeStep'>, 'straylight': <class 'jwst.straylight.straylight_step.StraylightStep'>, 'wavecorr': <class 'jwst.wavecorr.wavecorr_step.WavecorrStep'>, 'wfss_contam': <class 'jwst.wfss_contam.wfss_contam_step.WfssContamStep'>}

Methods Documentation

process(*data*)

Entrypoint for this pipeline

Parameters

input (*str* (https://docs.python.org/3/library/stdtypes.html#str), *Level2 Association*, *or JwstDataModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel.html#jwst.
– The exposure or association of exposures to process

process_exposure_product(*exp_product*, *pool_name=''*, *asn_file=''*)

Process an exposure found in the association product

Parameters

exp_product (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – A Level2b association product.

Spec3Pipeline

class jwst.pipeline.**Spec3Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

Spec3Pipeline: Processes JWST spectroscopic exposures from Level 2b to 3.

Included steps are: assign moving target wcs (assign_mtwcs) master background subtraction (master_background) MIRI MRS background matching (mrs_imatch) outlier detection (outlier_detection) 2-D spectroscopic resampling (resample_spec) 3-D spectroscopic resampling (cube_build) 1-D spectral extraction (extract_1d) Absolute Photometric Calibration (photom) 1-D spectral combination (combine_1d)

See Step.__init__ for the parameters.

Attributes Summary

class_alias	
spec	
step_defs	

Methods Summary

process(input)	Entrypoint for this pipeline

Attributes Documentation

class_alias = 'calwebb_spec3'

spec

step_defs = {'assign_mtwcs': <class 'jwst.assign_mtwcs.assign_mtwcs_step.AssignMTWcsStep'>, 'combine_1d': <class 'jwst.combine_1d.combine_1d_step.Combine1dStep'>, 'cube_build': <class 'jwst.cube_build.cube_build_step.CubeBuildStep'>, 'extract_1d': <class 'jwst.extract_1d.extract_1d_step.Extract1dStep'>, 'master_background': <class 'jwst.master_background.master_background_step.MasterBackgroundStep'>, 'mrs_imatch': <class 'jwst.mrs_imatch.mrs_imatch_step.MRSIMatchStep'>, 'outlier_detection': <class 'jwst.outlier_detection.outlier_detection_step.OutlierDetectionStep'>, 'photom': <class 'jwst.photom.photom_step.PhotomStep'>, 'resample_spec': <class 'jwst.resample.resample_spec_step.ResampleSpecStep'>, 'spectral_leak': <class 'jwst.spectral_leak.spectral_leak_step.SpectralLeakStep'>}

Methods Documentation

process(*input*)

Entrypoint for this pipeline

Parameters

input (*str* (https://docs.python.org/3/library/stdtypes.html#str), *Level3 Association*, *or JwstDataModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel.html#jwst.) – The exposure or association of exposures to process

Tso3Pipeline

class jwst.pipeline.**Tso3Pipeline**(**args*, ***kwargs*)

Bases: JwstPipeline

TSO3Pipeline: Applies level 3 processing to TSO-mode data from any JWST instrument.

Included steps are:

- outlier_detection
- tso_photometry
- extract_1d
- photom
- white_light

See Step.__init__ for the parameters.

Attributes Summary

class_alias	
reference_file_types	
spec	
step_defs	

Methods Summary

process(input)	Run the TSO3Pipeline

Attributes Documentation

class_alias = 'calwebb_tso3'

reference_file_types = ['gain', 'readnoise']

spec

```
scale_detection = boolean(default=False)
```

step_defs = {'extract_1d': <class 'jwst.extract_1d.extract_1d_step.Extract1dStep'>, 'outlier_detection': <class 'jwst.outlier_detection.outlier_detection_step.OutlierDetectionStep'>, 'photom': <class 'jwst.photom.photom_step.PhotomStep'>, 'tso_photometry': <class 'jwst.tso_photometry.tso_photometry_step.TSOPhotometryStep'>, 'white_light': <class 'jwst.white_light.white_light_step.WhiteLightStep'>}

Methods Documentation

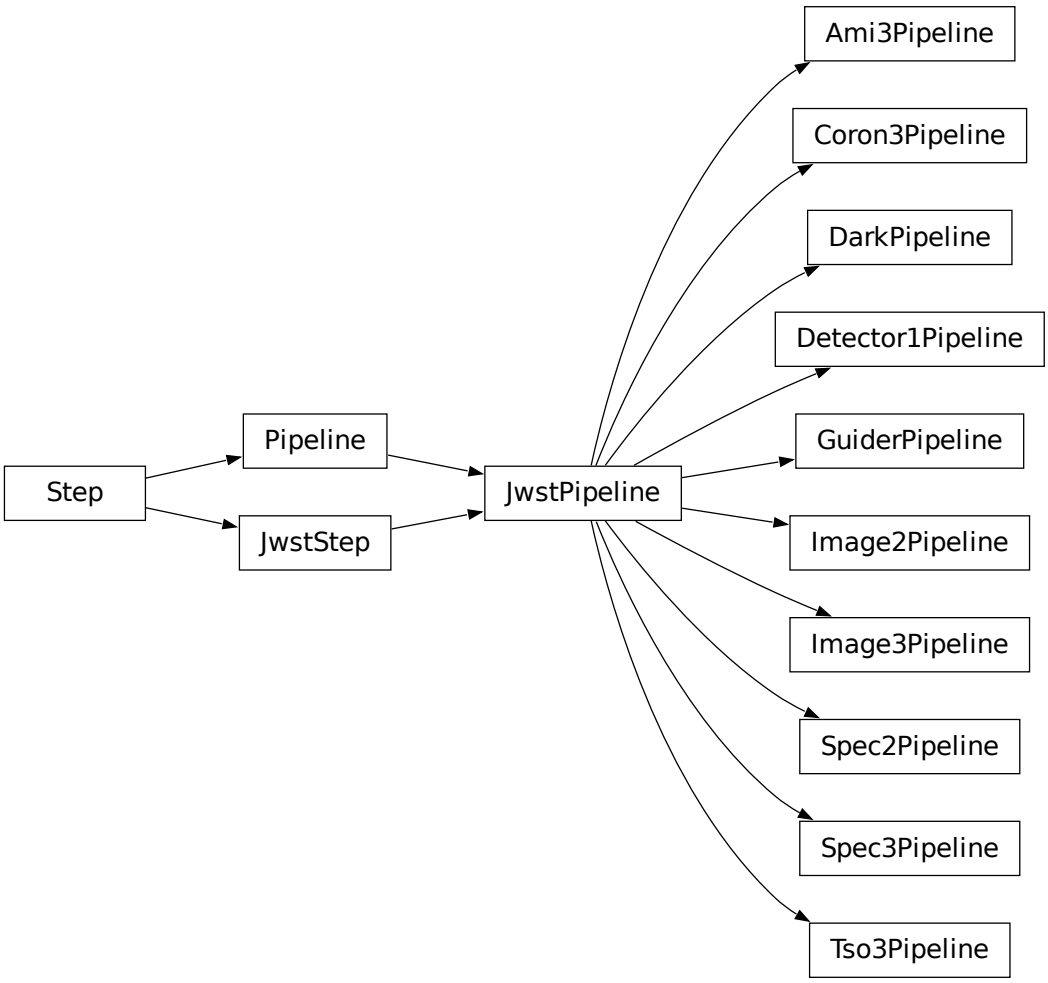
process(*input*)

Run the TSO3Pipeline

Parameters

input (*Level3 Association, json format*) – The exposures to process

Class Inheritance Diagram



15.1.45 Pixel Replacement

Description

Classes

jwst.pixel_replace.PixelReplaceStep

Alias

pixel_replace

During 1-D spectral extraction (*extract_1d* step), pixels flagged as bad are ignored in the summation process. If a bad pixel is part of the point-spread function (PSF) at a given wavelength, the absence of the signal in the flagged pixel will lead to a hollow space at that wavelength in the extracted spectrum.

To avoid this defect in the 1-D spectrum, this step estimates the flux values of pixels flagged as DO_NOT_USE in 2-D extracted spectra using interpolation methods, prior to rectification in the *resample_spec* step. pixel_replace inserts these estimates into the 2-D data array, unsets the DO_NOT_USE flag, and sets the FLUX_ESTIMATED flag for each affected pixel.

This step is provided as a cosmetic feature and, for that reason, should be used with caution.

Algorithms

Adjacent Profile Approximation

This is the default (and most extensively tested) algorithm for most spectroscopic modes.

First, the input 2-D spectral cutout is scanned across the dispersion axis to determine which cross-dispersion vectors (column or row, depending on dispersion direction) contain at least one flagged pixel. Next, for each affected vector, a median normalized profile is created.

The adjacent arrays (the number of which is set by the step argument n_adjacent_cols) are individually normalized. Next, each pixel in the profile is set to the median of the normalized values. This results in a median of normalized values filling the vector.

Finally, this profile is scaled to the vector containing a missing pixel, and the value is estimated from the scaled profile.

Minimum Gradient Estimator

In the case of the MIRI MRS, NaN-valued pixels are partially compensated during the IFU cube building process using the overlap between detector pixels and output cube voxels. The effects of NaN values are thus not as severe as for slit spectra, but can manifest as small dips in the extracted spectrum when a NaN value lands atop the peak of a spectral trace and cube building interpolates from lower-flux adjacent values.

Pixel replacement can thus be useful in some science cases for the MIRI MRS as well, but undersampling combined with the curvature of spectral traces on the detector can lead the model-based adjacent profile estimator to derive incorrect values in the vicinity of emission lines. The minimum gradient estimator is thus another optional algorithm that uses entirely local information to fill in the missing pixel values.

This method tests the gradient along the spatial and spectral axes using immediately adjacent pixels. It chooses whichever dimension has the minimum absolute gradient and replaces the missing pixel with the average of the two adjacent pixels along that dimension. Near point sources this will thus favor replacement along the spectral axis due to spatial undersampling of the PSF profile, while near bright extended emission lines it will favor replacement along the spatial axis due to the steep spectral profile. No replacement is attempted if a NaN value is bordered by another NaN value along a given axis.

Step Arguments

The pixel_replace step has the following step-specific arguments:

--algorithm (str, default='fit_profile')
This sets the method used to estimate flux values for bad pixels. The default 'fit_profile' uses a profile fit to adjacent column values. The minimum gradient ('mingrad') method is also available for the MIRI MRS.

--n_adjacent_cols (int, default=3)
Number of adjacent columns (on either side of column containing a bad pixel) to use in creation of the source profile, in cross-dispersion direction. The total number of columns used in the profile will be twice this number; on array edges, the total number of columns contributing to the source profile will be less than 2 * n_adjacent_cols.

Reference File

This step does not use any reference file.

jwst.pixel_replace Package

Classes

PixelReplaceStep([name, parent, ...])	PixelReplaceStep: Module for replacing flagged bad pixels with an estimate of their flux, prior to spectral extraction.

PixelReplaceStep

class jwst.pixel_replace.**PixelReplaceStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

PixelReplaceStep: Module for replacing flagged bad pixels with an estimate of their flux, prior to spectral extraction.

algorithm

Method used to estimate flux values for bad pixels. Currently only one option is implemented, using a profile fit to adjacent column values.

Type
str (https://docs.python.org/3/library/stdtypes.html#str)

n_adjacent_cols

Number of adjacent columns (on either side of column containing a bad pixel) to use in creation of source profile, in cross-dispersion direction. The total number of columns used in the profile will be twice this number; on array edges, take adjacent columns until this number is reached.

Type
int (https://docs.python.org/3/library/functions.html#int)

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	Execute the step.

Attributes Documentation

class_alias = 'pixel_replace'

spec

```
algorithm = option("fit_profile", "mingrad", "N/A", default="fit_profile")
n_adjacent_cols = integer(default=3)    # Number of adjacent columns to use in␣
↪creation of profile
skip = boolean(default=True) # Step must be turned on by parameter reference or␣
↪user
```

Methods Documentation

process(*input*)

Execute the step.

Parameters

input (*JWST DataModel*) –

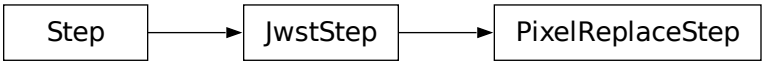
Returns

This will be `input` (https://docs.python.org/3/library/functions.html#input) if the step was skipped; otherwise, it will be a model containing data arrays with estimated fluxes for any bad pixels, now flagged as TO-BE-DETERMINED (DQ bit 7?).

Return type

JWST DataModel

Class Inheritance Diagram



15.1.46 Ramp Fitting

Description

Class
jwst.ramp_fitting.RampFitStep

Alias
ramp_fit

This step determines the mean count rate, in units of counts per second, for each pixel by performing a linear fit to the data in the input file. The fit is done using the "ordinary least squares" method. The fit is performed independently for each pixel. There can be up to three output files created by the step:

1. The primary output file ("rate") contains slope and other results at each pixel averaged over all integrations in the exposure.
2. The secondary product ("rateints") contains slope and other results for each integration, stored as data cubes.
3. A third, and optional, output product is also available, containing detailed fit information for each ramp segment for each pixel.

The three types of output products are described in more detail below.

The count rate for each pixel is determined by a linear fit to the cosmic-ray-free and saturation-free ramp intervals for each pixel. Hereafter such intervals will be referred to as a "segment." The fitting algorithm uses an 'optimal' weighting scheme, as described by Fixsen et al, PASP, 112, 1350. Details of the computations are given below.

Segments are determined using the 4-D GROUPDQ array of the input data set, under the assumption that the *saturation detection* and *jump detection* steps have already been applied, in order to flag occurrences of both saturation and cosmic-ray (CR) hits. Segments are terminated where saturation flags are found. Pixels are processed simultaneously in blocks using the array-based functionality of numpy. The size of the block depends on the image size and the number of groups per integration.

Upon successful completion of this step, the status keyword S_RAMP will be set to "COMPLETE".

Note that the core algorithms for this step are called from the external package `stcal`, an STScI effort to unify common calibration processing algorithms for use by multiple observatories.

Multiprocessing

This step has the option of running in multiprocessing mode. In that mode it will split the input data cube into a number of row slices based on the number of available cores on the host computer and the value of the max_cores input parameter. By default the step runs on a single processor. At the other extreme if max_cores is set to 'all', it will use all available cores (real and virtual). Testing has shown a reduction in the elapsed time for the step proportional to the number of real cores used. Using the virtual cores also reduces the elapsed time but at a slightly lower rate than the real cores.

Detailed Algorithms

Special Cases

If the input dataset has only a single group in each integration, the count rate for all unsaturated pixels in that integration will be calculated as the value of the science data in that group divided by the group time. If the input dataset has only two groups per integration, the count rate for all unsaturated pixels in each integration will be calculated using the differences between the two valid groups of the science data.

For datasets having more than a single group in each integration, a ramp having a segment with only a single group is processed differently depending on the number and size of the other segments in the ramp. If a ramp has only one segment and that segment contains a single group, the count rate will be calculated to be the value of the science data in that group divided by the group time. If a ramp has a segment having a single group, and at least one other segment having more than one good group, only data from the segment(s) having more than a single good group will be used to calculate the count rate.

The data are checked for ramps in which there is good data in the first group, but all first differences for the ramp are undefined because the remainder of the groups are either saturated or affected by cosmic rays. For such ramps, the first differences will be set to equal the data in the first group. The first difference is used to estimate the slope of the ramp, as explained in the 'segment-specific computations' section below.

If any input dataset contains ramps saturated in their second group, the count rates for those pixels in that integration will be calculated as the value of the science data in the first group divided by the group time.

MIRI First and Last Frames

The MIRI *first frame* correction step flags all pixels in the first group of each integration, so that those data do not get used in either the jump detection or ramp fitting steps. Similarly, the MIRI *last frame* correction step flags all pixels in the last group of each integration. The ramp fitting will only fit data if there are at least 2 good groups of data and will log a warning otherwise.

NIRCam Frame 0

If the input data contains a frame zero data cube, those data will be used to estimate a slope for pixels that are saturated in all groups. If all groups in an integration are flagged as SATURATED for a given pixel, the frame zero data array is examined to determine whether or not it is also saturated. Saturated elements of the frame zero array are set to zero by the preceding *saturation* step in the pipeline. Unsaturated elements will have non-zero values in the frame zero array. If the frame zero is not saturated, then it's value will be divided by the frame time for the exposure in order to compute a slope for the pixel in that integration. This is analagous to the situation in which only the first group in an integration is unsaturated and used by itself to compute a slope (see above).

Note that the computation of slopes from either a single group or single frame zero value is disabled when the step parameter `suppress_one_group` is set to `True`. In this case the slope value for a pixel with only one good sample will be set to zero.

All Cases

For all input datasets, including the special cases described above, arrays for the primary output (rate) product are computed as follows.

After computing the slopes for all segments for a given pixel, the final slope is determined as a weighted average from all segments in all integrations, and is written as the primary output product. In this output product, the 4-D GROUPDQ from all integrations is collapsed into 2-D, merged (using a bitwise OR) with the input 2-D PIXELDQ, and stored as a 2-D DQ array. The 3-D VAR_POISSON and VAR_RNOISE arrays from all integrations are averaged into corresponding 2-D output arrays. In cases where the median rate for a pixel is negative, the VAR_POISSON is set to zero, in order to avoid the unphysical situation of having a negative variance.

The slope images for each integration are stored as a data cube in a second output data product (rateints). Each plane of the 3-D SCI, ERR, DQ, VAR_POISSON, and VAR_RNOISE arrays in this product corresponds to the result for a given integration. In this output product, the GROUPDQ data for a given integration is collapsed into 2-D, which is then merged with the input 2-D PIXELDQ to create the output DQ array for each integration. The 3-D VAR_POISSON and VAR_RNOISE arrays are calculated by averaging over the fit segments in the corresponding 4-D variance arrays.

A third, optional output product is also available and is produced only when the step parameter 'save_opt' is True (the default is False). This optional product contains 4-D arrays called SLOPE, SIGSLOPE, YINT, SIGYINT, WEIGHTS, VAR_POISSON, and VAR_RNOISE that contain the slopes, uncertainties in the slopes, y-intercept, uncertainty in the y-intercept, fitting weights, the variance of the slope due to poisson noise only, and the variance of the slope due to read noise only for each segment of each pixel, respectively. The y-intercept refers to the result of the fit at an effective exposure time of zero. This product also contains a 3-D array called PEDESTAL, which gives the signal at zero exposure time for each pixel, and the 4-D CRMAG array, which contains the magnitude of each group that was flagged as having a CR hit. By default, the name of this output file will have the suffix "_fitopt". In this optional output product, the pedestal array is calculated for each integration by extrapolating the final slope (the weighted average of the slopes of all ramp segments in the integration) for each pixel from its value at the first group to an exposure time of zero. Any pixel that is saturated on the first group is given a pedestal value of 0. Before compression, the cosmic ray magnitude array is equivalent to the input SCI array but with the only nonzero values being those whose pixel locations are flagged in the input GROUPDQ as cosmic ray hits. The array is compressed, removing all groups in which all the values are 0 for pixels having at least one group with a non-zero magnitude. The order of the cosmic rays within the ramp is preserved.

Slope and Variance Calculations

Slopes and their variances are calculated for each segment, for each integration, and for the entire exposure. As defined above, a segment is a set of contiguous groups where none of the groups is saturated or cosmic-ray impacted. The appropriate slopes and variances are output to the primary output product, the integration-specific output product, and the optional output product. The following is a description of these computations. The notation in the equations is the following: the type of noise (when appropriate) will appear as the superscript 'R', 'P', or 'C' for readnoise, Poisson noise, or combined, respectively; and the form of the data will appear as the subscript: 's', 'i', 'o' for segment, integration, or overall (for the entire dataset), respectively.

Optimal Weighting Algorithm

The slope of each segment is calculated using the least-squares method with optimal weighting, as described by Fixsen et al. 2000, PASP, 112, 1350; Regan 2007, JWST-STScI-001212. Optimal weighting determines the relative weighting of each sample when calculating the least-squares fit to the ramp. When the data have low signal-to-noise ratio S, the data are read noise dominated and equal weighting of samples is the best approach. In the high signal-to-noise regime, data are Poisson-noise dominated and the least-squares fit is calculated with the first and last samples. In most practical cases, the data will fall somewhere in between, where the weighting is scaled between the two extremes.

The signal-to-noise ratio S used for weighting selection is calculated from the last sample as:

$$S = \frac{data \times gain}{\sqrt{(read\_noise)^2 + (data \times gain)}},$$

The weighting for a sample i is given as:

$$w_i = (i - i_{midpoint})^P,$$

where $i_{midpoint}$ is the the sample number of the midpoint of the sequence, and P is the exponent applied to weights, determined by the value of S. Fixsen et al. 2000 found that defining a small number of P values to apply to values of S was sufficient; they are given as:

Minimum S	Maximum S	P
0	5	0
5	10	0.4
10	20	1
20	50	3
50	100	6
100		10

Segment-specific Computations

The variance of the slope of a segment due to read noise is:

$$var_s^R = \frac{12\ R^2}{(ngroups_s^3 - ngroups_s)(tgroup^2)},$$

where R is the noise in the difference between 2 frames, $ngroups_s$ is the number of groups in the segment, and $tgroup$ is the group time in seconds (from the keyword TGROUP). The derivation of this equation is given in the appendix of this section, at readnoise variance derivation. (https://jwst-pipeline.readthedocs.io/en/latest/jwst/ramp_fitting/appendix.html).

The variance of the slope in a segment due to Poisson noise is:

$$var_s^P = \frac{slope_{est}}{tgroup \times gain\ (ngroups_s - 1)},$$

where $gain$ is the gain for the pixel (from the GAIN reference file), in e/DN. The $slope_{est}$ is an overall estimated slope of the pixel, calculated by taking the median of the first differences of the groups that are unaffected by saturation and cosmic rays, in all integrations. This is a more robust estimate of the slope than the segment-specific slope, which may be noisy for short segments.

The combined variance of the slope of a segment is the sum of the variances:

$$var_s^C = var_s^R + var_s^P$$

Integration-specific Computations

The variance of the slope for an integration due to read noise is:

$$var_i^R = \frac{1}{\sum_s \frac{1}{var_s^R}},$$

where the sum is over all segments in the integration.

The variance of the slope for an integration due to Poisson noise is:

$$var_i^P = \frac{1}{\sum_s \frac{1}{var_s^P}}$$

The combined variance of the slope for an integration due to both Poisson and read noise is:

$$var_i^C = \frac{1}{\sum_s \frac{1}{var_s^R + var_s^P}}$$

The slope for an integration depends on the slope and the combined variance of each segment's slope:

$$slope_i = \frac{\sum_s \frac{slope_s}{var_s^C}}{\sum_s \frac{1}{var_s^C}}$$

Exposure-level Computations

The variance of the slope due to read noise depends on a sum over all integrations:

$$var_o^R = \frac{1}{\sum_i \frac{1}{var_i^R}}$$

The variance of the slope due to Poisson noise is:

$$var_o^P = \frac{1}{\sum_i \frac{1}{var_i^P}}$$

The combined variance of the slope is the sum of the variances:

$$var_o^C = var_o^R + var_o^P$$

The square-root of the combined variance is stored in the ERR array of the primary output.

The overall slope depends on the slope and the combined variance of the slope of each integration's segments, so is a sum over integrations and segments:

$$slope_o = \frac{\sum_{i,s} \frac{slope_{i,s}}{var_{i,s}^C}}{\sum_{i,s} \frac{1}{var_{i,s}^C}}$$

Variances in Output Products

If the user requests creation of the optional output product, the variances of segment-specific slopes due to Poisson noise, var_s^P, and read noise, var_s^R, are stored in the VAR_POISSON and VAR_RNOISE file extensions, respectively.

At the integration-level, the variance of the per-integration slope due to Poisson noise, var_i^P, is written to the VAR_POISSON extension of the per-integration ("rateints") product, and the variance of the per-integration slope due to read noise, var_i^R, is written to the VAR_RNOISE extension. The square-root of the combined variance per integration due to both Poisson and read noise, var_i^C, is written to the ERR extension.

For the primary exposure-level ("rate") product, the overall variance in slope due to Poisson noise, var_o^P, is stored in the VAR_POISSON extension, the variance due to read noise, var_o^R, is stored in the VAR_RNOISE extension, and the square-root of the combined variance, var_o^C, is stored in the ERR extension.

Weighted Readnoise Variance

If the *charge migration* step has been performed prior to ramp fitting, any group whose value exceeds the `signal_threshold` parameter will have been flagged with the CHARGELOSS and DO_NOT_USE data quality flags. Due to the DO_NOT_USE flags, such groups will be excluded from the slope calculations.

However, it is desired to have a readnoise variance that is similar to pixels unaffected by charge migration, so an additional type of variance will be calculated, in which the excluded groups mentioned above will be included. This additional, 'weighted', readnoise variance is used for weighting in the *resample* step later in the pipeline. The 'weighted' readnoise variance is written to the VAR_RNOISE extension of each of the 3 output products.

The original ('conventional') type of readnoise variance described earlier is still used internally in other variance calculations but, as mentioned above, is no longer written to the separate variance extension.

Arguments

The ramp fitting step has three optional arguments that can be set by the user:

- `--save_opt`: A True/False value that specifies whether to write the optional output product. Default is False.
- `--opt_name`: A string that can be used to override the default name for the optional output product.
- `--int_name`: A string that can be used to override the default name for the per-integration product.
- `--suppress_one_group`: A boolean to suppress computations for saturated ramps with only one good (unsaturated) sample. The default is set to True to suppress these computations, which will compute all values for the ramp the same as if the entire ramp were saturated.
- `--maximum_cores`: The number of available cores that will be used for multi-processing in this step. The default value is '1' which does not use multi-processing. The other options are either an integer, 'quarter', 'half', and 'all'. Note that these fractions refer to the total available cores and on most CPUs these include physical and virtual cores. The clock time for the step is reduced almost linearly by the number of physical cores used on all machines. For example, on an Intel CPU with six real cores and six virtual cores, setting maximum_cores to 'half' results in a decrease of a factor of six in the clock time for the step to run. Depending on the system, the clock time can also decrease even more with maximum_cores is set to 'all'. Setting the number of cores to an integer can be useful when running on machines with a large number of cores where the user is limited in how many cores they can use.

Reference Files

The `ramp_fit` step uses two reference file types: GAIN and READNOISE. During ramp fitting, the GAIN values are used to temporarily convert the pixel values from units of DN to electrons, and convert the results of ramp fitting back to DN. The READNOISE values are used as part of the noise estimate for each pixel. Both are necessary for proper computation of noise estimates.

GAIN

READNOISE

Appendix

The derivation of the segment-specific readnoise variance (var_s^R) is shown here. This pertains to both the 'conventional' and 'weighted` readnoise variances - the only difference being the number of groups in the segment. This derivation follows the standard procedure to fitting data to a straight line, such as in chapter 15 of Numerical Recipes. The segment-specific variance from read noise corresponds to σ_b^2 in section 15.2.

For read noise R, weight w = $1/R^2$, which is a constant.

n = number of groups (`ngroups` in the text)

t = group time (`tgroup` in the text)

x = starting time for each group, = $(1, 2, 3, ...n + 1) \cdot t$

$S_1 = \sum_{k=1}^{n} w$

$S_x = \sum_{k=1}^{n} (w \cdot x_k)t$

$S_{xx} = \sum_{k=1}^{n} (w \cdot x_k)^2 t^2$

D = $S_1 \cdot S_{xx} - S_x^2$

Summations needed:

$\sum_{k=1}^{n} k = n \cdot (n + 1)/2 = n^2/2 + n/2$

$\sum_{k=1}^{n} k^2 = n \cdot (n + 1) \cdot (2 \cdot n + 1)/6 = n^3/3 + n^2/2 + n/6$

The variance from read noise = $var_s^R = S_1/D = S_1/(S_1 \cdot S_{xx} - S_x^2)$

$$= \frac{w \cdot n}{[w \cdot n \cdot \sum_{k=1}^{n} (w \cdot x_k^2 \cdot t^2)] - [\sum_{k=1}^{n} (w \cdot x_k \cdot t)]^2}$$

$$= \frac{n}{w \cdot t^2 \cdot [n \cdot (n^3/3 + n^2/2 + n/6) - (n^2/2 + n/2)^2]}$$

$$= \frac{1}{(n^3/12 - n/12) \cdot w \cdot t^2}$$

$$= \frac{12 \cdot R^2}{(n^3 - n) \cdot t^2}$$

This is the equation in the code and in the segment-specific computations section of the Description.

jwst.ramp_fitting Package

Classes

RampFitStep([name, parent, config_file, ...])	This step fits a straight line to the value of counts vs.

RampFitStep

class jwst.ramp_fitting.**RampFitStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

This step fits a straight line to the value of counts vs. time to determine the mean count rate for each pixel.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

algorithm
class_alias
reference_file_types
spec
weighting

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

algorithm = 'ols'

class_alias = 'ramp_fit'

reference_file_types = ['readnoise', 'gain']

spec

```
int_name = string(default='')
save_opt = boolean(default=False) # Save optional output
opt_name = string(default='')
suppress_one_group = boolean(default=True)  # Suppress saturated ramps with␣
↪good 0th group
maximum_cores = string(default='1') # cores for multiprocessing. Can be an␣
↪integer, 'half', 'quarter', or 'all'
```

weighting = 'optimal'

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.47 Reference File Information

Introduction

This document is intended to be a core reference guide to the formats, naming convention and data quality flags used by the reference files for pipeline steps requiring them, and is not intended to be a detailed description of each of those pipeline steps. It also does not give details on pipeline steps that do not use reference files. The present manual is referred to by several other documentation pages, such as the JWST pipeline and JDocs.

Reference File Naming Convention

Before reference files are ingested into CRDS, they are renamed following a convention used by the pipeline. As with any other changes undergone by the reference files, the previous names are kept in header keywords, so the Instrument Teams can easily track which delivered file is being used by the pipeline in each step.

The naming of reference files uses the following syntax:

```
jwst_<instrument>_<reftype>_<version>.<extension>
```

where

- `instrument` is one of "fgs", "miri", "nircam", "niriss", and "nirspec"
- `reftype` is one of the type names listed in the table below
- `version` is a 4-digit version number (e.g. 0042)
- `extension` gives the file format, such as "fits" or "asdf"

An example NIRCam GAIN reference file name would be "jwst_nircam_gain_0042.fits".

The HISTORY header keyword of each reference file includes details on specific processing undergone by the files before being ingested in CRDS.

Reference File Types

Most reference files have a one-to-one relationship with calibration steps, e.g. there is one step that uses one type of reference file. Some steps, however, use several types of reference files and some reference file types are used by more than one step. The tables below show the correspondence between pipeline steps and reference file types. The first table is ordered by pipeline step, while the second is ordered by reference file type. Links to the reference file types provide detailed documentation on each reference file.

Pipeline Step	Reference File Type (REFTYPE)
align_refs	*PSFMASK*
ami_analyze	*THROUGHPUT*
assign_wcs	CAMERA
	COLLIMATOR
	DISPERSER
	DISTORTION
	FILTEROFFSET
	FORE
	FPA
	IFUFORE
	IFUPOST

continues on next page

Table 4 – continued from previous page

Pipeline Step	Reference File Type (REFTYPE)
	IFUSLICER
	MSA
	OTE
	SPECWCS
	REGIONS
	WAVELENGTHRANGE
background	*WFSSBKG*
	WAVELENGTHRANGE
barshadow	*BARSHADOW*
cube_build	*CUBEPAR*
dark_current	*DARK*
dq_init	*MASK*
emicorr	*EMICORR*
extract_1d	*EXTRACT1D*
	APCORR
	SPECKERNEL (NIRISS SOSS ATOCA only)
	SPECPROFILE (NIRISS SOSS ATOCA only)
	SPECTRACE (NIRISS SOSS ATOCA only)
	WAVEMAP (NIRISS SOSS ATOCA only)
extract_2d	*WAVECORR*
	WAVELENGTHRANGE
flatfield	*FLAT*
	DFLAT
	FFLAT
	SFLAT
fringe	*FRINGE*
gain_scale	GAIN
ipc	*IPC*
jump	GAIN
	READNOISE
linearity	*LINEARITY*
msaflagopen	*MSAOPER*
pathloss	*PATHLOSS*
persistence	*PERSAT*
	TRAPDENSITY
	TRAPPARS
photom	*PHOTOM*
	AREA
ramp_fitting	GAIN
	READNOISE
refpix	*REFPIX*
resample	*DRIZPARS*
reset	*RESET*
residual_fringe	*FRINGEFREQ*
	REGIONS
rscd	*RSCD*
saturation	*SATURATION*
source_catalog	APCORR
	ABVEGAOFFSET
straylight	*MRSXARTCORR*

continues on next page

Table 4 – continued from previous page

Pipeline Step	Reference File Type (REFTYPE)
spectral_leak	*MRSPTCORR*
superbias	*SUPERBIAS*
tso_photometry	*TSOPHOT*
wavecorr	*WAVECORR*

Reference File Type (REFTYPE)	Pipeline Step
ABVEGAOFFSET	*source_catalog*
APCORR	*extract_1d*
	source_catalog
AREA	*photom*
BARSHADOW	*barshadow*
CAMERA	*assign_wcs*
COLLIMATOR	*assign_wcs*
CUBEPAR	*cube_build*
DARK	*dark_current*
DFLAT	*flatfield*
DISPERSER	*assign_wcs*
DISTORTION	*assign_wcs*
DRIZPARS	*resample*
EMICORR	*emicorr*
EXTRACT1D	*extract_1d*
FFLAT	*flatfield*
FILTEROFFSET	*assign_wcs*
FLAT	*flatfield*
FORE	*assign_wcs*
FPA	*assign_wcs*
FRINGE	*fringe*
FRINGEFREQ	*residual_fringe*
GAIN	*gain_scale*
	jump
	ramp_fitting
IFUFORE	*assign_wcs*
IFUPOST	*assign_wcs*
IFUSLICER	*assign_wcs*
IPC	*ipc*
LINEARITY	*linearity*
MASK	*dq_init*
MRSPTCORR	*spectral_leak*
MRSXARTCORR	*straylight*
MSA	*assign_wcs*
MSAOPER	*msaflagopen*
OTE	*assign_wcs*
PATHLOSS	*pathloss*
PERSAT	*persistence*
PHOTOM	*photom*
PSFMASK	*align_refs*
READNOISE	*jump*
	ramp_fitting

continues on next page

Table 5 – continued from previous page

Reference File Type (REFTYPE)	Pipeline Step
REFPIX	*refpix*
REGIONS	*assign_wcs*
	residual_fringe
RESET	*reset*
RSCD	*rscd*
SATURATION	*saturation*
SFLAT	*flatfield*
SPECWCS	*assign_wcs*
SUPERBIAS	*superbias*
THROUGHPUT	*ami_analyze*
TRAPDENSITY	*persistence*
TRAPPARS	*persistence*
TSOPHOT	*tso_photometry*
WAVELENGTHRANGE	*assign_wcs*
	background
	extract_2d
WAVECORR	*wavecorr*
WFSSBKG	*background*

Step Parameters Reference Types

When each `Step` is instantiated, a CRDS look-up, based on the `Step` class name and input data, is made to retrieve a parameter file. The `reftype` for such parameter files is `pars-<class name>`. For example, for the step `jwst.persistence.PersistenceStep`, the `reftype` would be `pars-persistencestep`.

For more information, see *Parameter Files*.

Standard Required Keywords

At present, most JWST science and reference files are FITS files with image or table extensions. The FITS primary data unit is always empty. The primary header contains all keywords not specific to individual extensions. Keywords specific to a particular extension are contained in the header of that extension.

The required Keywords Documenting Contents of Reference Files are:

Keyword	Comment
REFTY	WFSSBKG Required values are listed in the discussion of each pipeline step.
DESCRIP	Summary of file content and/or reason for delivery
AUTHOR	Fred Jones Person(s) who created the file
USEAFTER	YYYY-MM-DDThh:mm:ss Date and time after the reference files will be used. The T is required. Time string may NOT be omitted; use T00:00:00 if no meaningful value is available.
PEDIGREE	Options are 'SIMULATION' 'GROUND' 'DUMMY' 'INFLIGHT YYYY-MM-DD YYYY-MM-DD'
HISTORY	Description of Reference File Creation
HISTORY	DOCUMENT: Name of document describing the strategy and algorithms used to create file.
HISTORY	SOFTWARE: Description, version number, location of software used to create file.
HISTORY	DATA USED: Data used to create file
HISTORY	DIFFERENCES: How is this version different from the one that it replaces?
HISTORY	If your text spills over to the next line, begin it with another HISTORY keyword, as in this example.
TELESCOP	JWST Name of the telescope/project.
INSTRUM	FGS Instrument name. Allowed values: FGS, NIRCAM, NIRISS, NIRSPEC, MIRI
SUBARRAY	FULL, GENERIC, SUBS200A1, ... (XXX abstract technical description of SUBARRAY)
SUBSTRT1	1 Starting pixel index along axis 1 (1-indexed)
SUBSIZE1	2048 Size of subarray along axis 1
SUBSTRT2	1 Starting pixel index along axis 2 (1-indexed)
SUBSIZE2	2048 Size of subarray along axis 2
FASTAXIS	1 Fast readout direction relative to image axes for Amplifier #1 (1 = +x axis, 2 = +y axis, -1 = -x axis, -2 = -y axis) SEE NOTE BELOW.
SLOWA	2 Slow readout direction relative to image axes for all amplifiers (1 = +x axis, 2 = +y axis, -1 = -x axis, -2 = -y axis)

Observing Mode Keywords

A pipeline module may require separate reference files for each instrument, detector, filter, observation date, etc. The values of these parameters must be included in the reference file header. The observing-mode keyword values are vital to the process of ingesting reference files into CRDS, as they are used to establish the mapping between observing modes and specific reference files. Some observing-mode keywords are also used in the pipeline processing steps. If an observing-mode keyword is irrelevant to a particular observing mode (such as GRATING for the MIRI imager mode or the NIRCam and NIRISS instruments), then it may be omitted from the file header.

The Keywords Documenting the Observing Mode are:

Keyword	Sample Value	Comment
PUPIL	NRM	Pupil wheel element. Required only for NIRCam and NIRISS. NIRCam allowed values: CLEAR, F162M, F164N, F323N, F405N, F466N, F470N, GRISMV2, GRISMV3 NIRISS allowed values: CLEARP, F090W, F115W, F140M, F150W, F158M, F200W, GR700XD, NRM
FILTER	F210	Filter wheel element. Allowed values: too many to list here
GRATING	G395	Required only for NIRSpec. NIRSpec allowed values: G140M, G235M, G395M, G140H, G235H, G395H, PRISM, MIRROR
EXP_	MIR_	Exposure type. FGS allowed values: FGS_IMAGE, FGS_FOCUS, FGS_SKYFLAT, FGS_INTFLAT, FGS_DARK MIRI allowed values: MIR_IMAGE, MIR_TACQ, MIR_LYOT, MIR_4QPM, MIR_LRS-FIXEDSLIT, MIR_LRS-SLITLESS, MIR_MRS, MIR_DARK, MIR_FLATIMAGE, MIR_FLATMRS, MIR_CORONCAL NIRCam allowed values: NRC_IMAGE, NRC_GRISM, NRC_TACQ, NRC_TACONFIRM, NRC_CORON, NRC_TSIMAGE, NRC_TSGRISM, NRC_FOCUS, NRC_DARK, NRC_FLAT, NRC_LED NIRISS allowed values: NIS_IMAGE, NIS_TACQ, NIS_TACONFIRM, NIS_WFSS, NIS_SOSS, NIS_AMI, NIS_FOCUS, NIS_DARK, NIS_LAMP NIRSpec allowed values: NRS_TASLIT, NRS_TACQ, NRS_TACONFIRM, NRS_CONFIRM, NRS_FIXEDSLIT, NRS_AUTOWAVE, NRS_IFU, NRS_MSASPEC, NRS_AUTOFLAT, NRS_IMAGE, NRS_FOCUS, NRS_DARK, NRS_LAMP, NRS_BOTA, NRS_BRIGHTOBJ, NRS_MIMF
DETECTOR	MIR-I-FULON	Allowed values: GUIDER1, GUIDER2 NIS NRCA1, NRCA2, NRCA3, NRCA4, NRCB1, NRCB2, NRCB3, NRCB4, NRCALONG, NRCBLONG NRS1, NRS2 MIRIFULONG, MIRIFUSHORT, MIRIMAGE
CHANNEL	12	MIRI MRS (IFU) channel. Allowed values: 1, 2, 3, 4, 12, 34 SHORT NIRCam channel. Allowed values: SHORT, LONG
BAND	MED	IFU band. Required only for MIRI. Allowed values are SHORT, MEDIUM, LONG, and N/A, as well as any allowable combination of two values (SHORT-MEDIUM, LONG-SHORT, etc.). (Also used as a header keyword for selection of all MIRI Flat files, Imager included.)
READPATT	FAST	Name of the readout pattern used for the exposure. Each pattern represents a particular combination of parameters like nframes and groups. For MIRI, FAST and SLOW refer to the rate at which the detector is read. MIRI allowed values: SLOW, FAST, FASTGRPAVG, FASTINTAVG NIRCam allowed values: DEEP8, DEEP2, MEDIUM8, MEDIUM2, SHALLOW4, SHALLOW2, BRIGHT2, BRIGHT1, RAPID NIRSpec allowed values: NRSRAPID, NRS, NRSN16R4, NRSIRS2RAPID NIRISS allowed values: NIS, NISRAPID FGS allowed values: ID, ACQ1, ACQ2, TRACK, FINEGUIDE, FGS60, FGS840, FGS7850, FGSRAPID, FGS
NRS_	16	Required only for NIRSpec.
NRS_	4	Required only for NIRSpec.
P_X?	P_RE	pattern keywords used by CRDS for JWST to describe the intended uses of a reference file using or'ed combinations of values. Only a subset of *P_pattern keywords* are supported.

Note: For the NIR detectors, the fast readout direction changes sign from one amplifier to the next. It is +1, -1, +1, and -1, for amps 1, 2, 3, and 4, respectively. The keyword FASTAXIS refers specifically to amp 1. That way, it is entirely

correct for single-amp readouts and correct at the origin for 4-amp readouts. For MIRI, FASTAXIS is always +1.

Tracking Pipeline Progress

As each pipeline step is applied to a science data product, it will record a status indicator in a header keyword of the science data product. The current list of step status keyword names is given in the following table. These status keywords may be included in the primary header of reference files, in order to maintain a history of the data that went into creating the reference file. Allowed values for the status keywords are 'COMPLETE' and 'SKIPPED'. Absence of a particular keyword is understood to mean that step was not even attempted.

Table 1. Keywords Documenting Which Pipeline Steps Have Been Performed.

S_AMIANA	AMI fringe analysis
S_AMIAVG	AMI fringe averaging
S_AMINOR	AMI fringe normalization
S_BARSHA	Bar shadow correction
S_BKDSUB	Background subtraction
S_COMB1D	1-D spectral combination
S_DARK	Dark subtraction
S_DQINIT	DQ initialization
S_EXTR1D	1-D spectral extraction
S_EXTR2D	2-D spectral extraction
S_FLAT	Flat field correction
S_FRINGE	Fringe correction
S_FRSTFR	MIRI first frame correction
S_GANSCL	Gain scale correction
S_GRPSCL	Group scale correction
S_GUICDS	Guide mode CDS computation
S_IFUCUB	IFU cube creation
S_IMPRNT	NIRSpec MSA imprint subtraction
S_IPC	IPC correction
S_JUMP	Jump detection
S_KLIP	Coronagraphic PSF subtraction
S_LASTFR	MIRI last frame correction
S_LINEAR	Linearity correction
S_MIREMI	MIRI EMI correction
S_MRSMAT	MIRI MRS background matching
S_MSAFLG	NIRSpec MSA failed shutter flagging
S_OUTLIR	Outlier detection
S_PERSIS	Persistence correction
S_PHOTOM	Photometric (absolute flux) calibration
S_PSFALI	Coronagraphic PSF alignment
S_PSFSTK	Coronagraphic PSF stacking
S_PTHLOS	Pathloss correction
S_RAMP	Ramp fitting
S_REFPIX	Reference pixel correction
S_RESAMP	Resampling (drizzling)
S_RESET	MIRI reset correction
S_RSCD	MIRI RSCD correction
S_SATURA	Saturation check
S_SKYMAT	Sky matching
S_SRCCAT	Source catalog creation

continues on next page

Table 6 – continued from previous page

S_SRCTYP	Source type determination
S_STRAY	Straylight correction
S_SUPERB	Superbias subtraction
S_TELEMI	Telescope emission correction
S_TSPHOT	TSO imaging photometry
S_TWKREG	Tweakreg image alignment
S_WCS	WCS assignment
S_WFSCOM	Wavefront sensing image combination
S_WHTLIT	TSO white-light curve generation

Orientation of Detector Image

All steps in the pipeline assume the data are in the DMS (science) orientation, not the native readout orientation. The pipeline does NOT check or correct for the orientation of the reference data. It assumes that all files ingested into CRDS have been put into the science orientation. All header keywords documenting the observing mode (Table 2) should likewise be transformed into the DMS orientation. For square data array dimensions it's not possible to infer the actual orientation directly so reference file authors must manage orientation carefully.

Table 2. Correct values for FASTAXIS and SLOWAXIS for each detector.

DETECTOR	FASTAXIS	SLOWAXIS
MIRIMAGE	1	2
MIRIFULONG	1	2
MIRIFUSHORT	1	2
NRCA1	-1	2
NRCA2	1	-2
NRCA3	-1	2
NRCA4	1	-2
NRCALONG	-1	2
NRCB1	1	-2
NRCB2	-1	2
NRCB3	1	-2
NRCB4	-1	2
NRCBLONG	1	-2
NRS1	2	1
NRS2	-2	-1
NIS	-2	-1
GUIDER1	-2	-1
GUIDER2	2	-1

Differing values for these keywords will be taken as an indicator that neither the keyword value nor the array orientation are correct.

P_pattern keywords

P_ pattern keywords used by CRDS for JWST to describe the intended uses of a reference file using or'ed combinations

For example, if the same NIRISS SUPERBIAS should be used for

READPATT=NIS

or

READPATT=NISRAPID

the definition of READPATT in the calibration s/w datamodels schema does not allow it. READPATT can specify one or the other but not both.

To support expressing combinations of values, CRDS and the CAL s/w have added "pattern keywords" which nominally begin with P_ followed by the ordinary keyword, truncated as needed to 8 characters. In this case, P_READPA corresponds to READPATT.

Pattern keywords override the corresponding ordinary keyword for the purposes of automatically updating CRDS rmaps. Pattern keywords describe intended use.

In this example, the pattern keyword:

P_READPA = NIS | NISRAPID |

can be used to specify the intent "use for NIS or for NISRAPID".

Only or-ed combinations of the values used in ordinary keywords are valid for pattern keywords.

Patterns appear in a slightly different form in rmaps than they do in P_ keywords. The value of a `P_ keyword` always ends with a trailing or-bar. In rmaps, no trailing or-bar is used so the equivalent of the above in an rmap is:

'NIS|NISRAPID'

From a CRDS perspective, the `P_ pattern` keywords and their corresponding datamodels paths currently supported can be found in the JWST Pattern Keywords section of the CRDS documentation. (https://jwst-crds.stsci.edu/static/users_guide/reference_conventions.html#id2)

Currently all P_ keywords correspond to basic keywords found only in the primary headers of reference files and are typically only valid for FITS format..

The translation from these P_ pattern keywords are completely generic in CRDS and can apply to any reference file type so they should be assumed to be reserved whether a particular type uses them or not. Defining non-pattern keywords with the prefix P_ is strongly discouraged.

Data Quality Flags

Within science data files, the PIXELDQ flags are stored as 32-bit integers; the GROUPDQ flags are 8-bit integers. The meaning of each bit is specified in a separate binary table extension called DQ_DEF. The binary table has the format presented in Table 3, which represents the master list of DQ flags. Only the first eight entries in the table below are relevant to the GROUPDQ array. All calibrated data from a particular instrument and observing mode have the same set of DQ flags in the same (bit) order. For Build 7, this master list will be used to impose this uniformity. We may eventually use different master lists for different instruments or observing modes.

Within reference files for some steps, the Data Quality arrays for some steps are stored as 8-bit integers to conserve memory. Only the flags actually used by a reference file are included in its DQ array. The meaning of each bit in the DQ array is stored in the DQ_DEF extension, which is a binary table having the following fields: Bit, Value, Name, and Description.

Table 3. Flags for the DQ, PIXELDQ, and GROUPDQ Arrays (Format of DQ_DEF Extension).

Bit	Value	Name	Description
0	1	DO_NOT_USE	Bad pixel. Do not use.
1	2	SATURATED	Pixel saturated during exposure
2	4	JUMP_DET	Jump detected during exposure
3	8	DROPOUT	Data lost in transmission
4	16	OUTLIER	Flagged by outlier detection
5	32	PERSISTENCE	High persistence
6	64	AD_FLOOR	Below A/D floor
7	128	CHARGELOSS	Charge Migration
8	256	UNRELIABLE_ERROR	Uncertainty exceeds quoted error
9	512	NON_SCIENCE	Pixel not on science portion of detector
10	1024	DEAD	Dead pixel
11	2048	HOT	Hot pixel
12	4096	WARM	Warm pixel
13	8192	LOW_QE	Low quantum efficiency
14	16384	RC	RC pixel
15	32768	TELEGRAPH	Telegraph pixel
16	65536	NONLINEAR	Pixel highly nonlinear
17	131072	BAD_REF_PIXEL	Reference pixel cannot be used
18	262144	NO_FLAT_FIELD	Flat field cannot be measured
19	524288	NO_GAIN_VALUE	Gain cannot be measured
20	1048576	NO_LIN_CORR	Linearity correction not available
21	2097152	NO_SAT_CHECK	Saturation check not available
22	4194304	UNRELIABLE_BIAS	Bias variance large
23	8388608	UNRELIABLE_DARK	Dark variance large
24	16777216	UNRELIABLE_SLOPE	Slope variance large (i.e., noisy pixel)
25	33554432	UNRELIABLE_FLAT	Flat variance large
26	67108864	OPEN	Open pixel (counts move to adjacent pixels)
27	134217728	ADJ_OPEN	Adjacent to open pixel
28	268435456	FLUX_ESTIMATED	Pixel flux estimated due to missing/bad data
29	536870912	MSA_FAILED_OPEN	Pixel sees light from failed-open shutter
30	1073741824	OTHER_BAD_PIXEL	A catch-all flag
31	2147483648	REFERENCE_PIXEL	Pixel is a reference pixel

Note: Words like “highly” and “large” will be defined by each instrument team. They are likely to vary from one detector to another – or even from one observing mode to another.

Parameter Specification

There are a number of steps, such as *OutlierDetectionStep* or *SkyMatchStep*, that define what data quality flags a pixel is allowed to have to be considered in calculations. Such parameters can be set in a number of ways.

First, the flag can be defined as the integer sum of all the DQ bit values from the input images DQ arrays that should be considered “good”. For example, if pixels in the DQ array can have combinations of 1, 2, 4, and 8 and one wants to consider DQ flags 2 and 4 as being acceptable for computations, then the parameter value should be set to “6” (2+4). In this case a pixel having DQ values 2, 4, or 6 will be considered a good pixel, while a pixel with a DQ value, e.g., 1+2=3, 4+8=”12”, etc. will be flagged as a “bad” pixel.

Alternatively, one can enter a comma-separated or ‘+’ separated list of integer bit flags that should be summed to obtain the final “good” bits. For example, both “4,8” and “4+8” are equivalent to a setting of “12”.

Finally, instead of integers, the JWST mnemonics, as defined above, may be used. For example, all the following specifications are equivalent:

```
"12" == "4+8" == "4, 8" == "JUMP_DET, DROPOUT"
```

Note: The default value (0) will make *all* non-zero pixels in the DQ mask be considered "bad" pixels and the corresponding pixels will not be used in computations.

Setting to `None` (https://docs.python.org/3/library/constants.html#None) will turn off the use of the DQ array for computations.

In order to reverse the meaning of the flags from indicating values of the "good" DQ flags to indicating the "bad" DQ flags, prepend '~' to the string value. For example, in order to exclude pixels with DQ flags 4 and 8 for computations and to consider as "good" all other pixels (regardless of their DQ flag), use a value of `~4+8`, or `~4,8`. A string value of `~0` would be equivalent to a setting of `None`.

CRDS Integration in CAL Code

For JWST, the Calibration Reference Data System (CRDS) is directly integrated with calibration steps and pipelines resulting in conventions for how Steps and Pipelines should be written.

Step Attribute .reference_file_types

Each calibration Step is required to define an attribute or property named *reference_file_types* which defines the CRDS reference file types that are required for running the calibration step. Note that for some Steps the reference file types actually used vary so the minimal list of required types may not be known if no science data is defined.

Note that the `Step` parameter reference files do not need to be specified. These are automatically requested in the `Step` architecture.

CRDS Prefetch

To ensure all reference files required by a pipeline are available prior to processing, Pipelines perform a "pre-fetch" of all references required by any Step in the Pipeline. This generic Pipeline behavior is intended to prevent processing which fails due to missing CRDS files after running for lengthy periods.

When *CRDS_SERVER_URL* and *CRDS_PATH* are properly configured, CRDS will download and locally cache the minimal set of references required to calibrate specific data using a particular pipeline. This configuration supports remote processing for users with no access or inefficient access to the STScI local network.

The pre-fetch also enables CRDS to report on all reference file assignment and availability problems a pipeline will encounter in a single CAL run. This is required in I&T scenarios where the total number of pipeline runs is very limited (often weekly) so solving as many reference file issues per run as possible is needed.

While the prefetch runs for onsite users, since the default CRDS configuration points to a complete CRDS cache, no downloads will occur.

Step Method .get_reference_file()

During processing individual Steps make secondary calls to CRDS via the *get_reference_file(input_file, reference_file_type)* method to fetch the cache paths of individual reference files. If no file is applicable, CRDS returns the value 'N/A' which is sometimes used to skip related Step processing. If there is an error determining a reference file, CRDS will raise an exception stopping the calibration; typically these occur due to missing reference files or incorrectly specified dataset parameters.

While get_reference_file() returns the absolute path of CRDS reference files, reference file assignments are recorded in output products using a *crds://* URI prefix which translates to roughly "the path of this file in the local cache you've defined using CRDS_PATH, or under /grp/crds/cache if you didn't define CRDS_PATH".

Best Reference Matching

The Calibration Reference Data System (CRDS) assigns the best reference files needed to process a data set based on the dataset's metadata (FITS headers) and plain text CRDS rules files.

CRDS rules and references are organized into a 4 tiered hierarchical network of versioned files consisting of:

- .pmap - The overall context for the pipeline (i.e. all instruments)
- .imap - The rules for all reference types of one instrument
- .rmap - The rules for all reference files of one type of one instrument
- .fits,.asdf,.json - Individual reference files assigned by .rmaps

Based on dataset parameters, CRDS traverses the hierarchy of rules files, generally starting from the .pmap and descending until a particular reference file is assigned.

Visiting the JWST operational website here:

https://jwst-crds.stsci.edu/

and opening up instrument panes of the *Operational References* display can rapidly give an idea about how reference files should be assigned.

CRDS Parameter Naming

For the sake of brevity, the CRDS website often refers to matching parameters using truncated names intended to give the gist of a parameter.

Within CRDS rules for JWST, CRDS refers to parameters using jwst datamodels attribute paths converted to capitalized strings analogous to FITS keywords. For instance the datamodels attribute:

```
meta.instrument.name
```

corresponds to CRDS rules parameter name:

```
'META.INSTRUMENT.NAME'
```

and FITS keyword:

```
'INSTRUME'
```

Using e.g. 'META.INSTRUMENT.NAME' permits consistent naming regardless of the underlying file format (.fits vs. .asdf vs. .json).

When creating or accessing reference files, Python code uses the lower case object path to populate an attribute corresponding to the upper case string.

Example .pmap contents

Generally CRDS reference file lookups begin with a .pmap (context) file.

The .pmap's serial number describes the overall version of rules for a pipeline.

The contents of context jwst_0493.pmap are shown below:

```
 header = {
    'mapping' : 'PIPELINE',
    'observatory' : 'JWST',
    'name' : 'jwst_0493.pmap',
    'parkey' : ('META.INSTRUMENT.NAME',),
    ...
}

selector = {
    'FGS' : 'jwst_fgs_0073.imap',
    'MIRI' : 'jwst_miri_0158.imap',
    'NIRCAM' : 'jwst_nircam_0112.imap',
    'NIRISS' : 'jwst_niriss_0117.imap',
    'NIRSPEC' : 'jwst_nirspec_0173.imap',
    'SYSTEM' : 'jwst_system_0017.imap',
}
```

Based on the parameter META.INSTRUMENT.NAME (INSTRUME) CRDS selects an appropriate .imap for further searching.

In all CRDS rules files, the header's **parkey** field defines the parameter names used to select a file. These parkey names correspond to the values shown in the selector's keys.

Conceptually all CRDS selectors consist of dictionaries which map parameter values to either a file or a sub-selector.

If META.INSTRUMENT.NAME=NIRSPEC, then CRDS would choose *jwst_nirspec_0173.imap* to continue it's search.

Example .imap contents

A .imap file defines the appropriate version of .rmap to search for each reference type supported by the corresponding instrument. Below is an example .imap taken from NIRSPEC:

```
header = {
  'mapping' : 'INSTRUMENT',
  'instrument' : 'NIRSPEC',
  'name' : 'jwst_nirspec_0173.imap',
  'parkey' : ('REFTYPE',),
  ...
}

selector = {
  'AREA' : 'jwst_nirspec_area_0010.rmap',
```

(continues on next page)

(continued from previous page)

```
  'BARSHADOW' : 'jwst_nirspec_barshadow_0002.rmap',
  'CAMERA' : 'jwst_nirspec_camera_0015.rmap',
  ...,
  'PATHLOSS' : 'jwst_nirspec_pathloss_0003.rmap',
  ...,
  'WAVECORR' : 'jwst_nirspec_wavecorr_0003.rmap',
  'WAVELENGTHRANGE' : 'jwst_nirspec_wavelengthrange_0015.rmap',
  'WCSREGIONS' : 'N/A',
  'WFSSBKG' : 'N/A',
}
```

A value of N/A indicates that a particular reference type is not yet used by this instrument and CRDS will return 'N/A' instead of a filename.

If the requested REFTYPE was PATHLOSS, CRDS would continue it's search with *jwst_nirspec_pathloss_0003.rmap*.

Example .rmap contents

Slightly modified contents of *jwst_nirspec_pathloss_0003.rmap* are shown below:

```
header = {
   'mapping' : 'REFERENCE',
   'observatory' : 'JWST',
   'instrument' : 'NIRSPEC',
   'filekind' : 'PATHLOSS',
   'name' : 'jwst_nirspec_pathloss_0003.rmap',
   'classes' : ('Match', 'UseAfter'),
   'parkey' : (('META.EXPOSURE.TYPE',), ('META.OBSERVATION.DATE', 'META.OBSERVATION.TIME
↪')),
   ...
}

selector = Match({
   'NRS_AUTOWAVE' : 'N/A',
   'NRS_FIXEDSLIT|NRS_BRIGHTOBJ' : UseAfter({
       '1900-01-01 00:00:00' : 'jwst_nirspec_pathloss_0001.fits',
   }),
   'NRS_IFU' : UseAfter({
       '1900-01-01 00:00:00' : 'jwst_nirspec_pathloss_0003.fits',
   }),
   'NRS_MSASPEC' : UseAfter({
       '1900-01-01 00:00:00' : 'jwst_nirspec_pathloss_0002.fits',
       '2000-01-01 00:00:00' : 'jwst_nirspec_pathloss_0007.fits',
   }),
})
```

Each class of CRDS rmap selector defines a search algorithm to be used at that stage of the reference file lookup.

Match Selector

In the example shown above, CRDS selects a nested UseAfter selector based on the value of META.EXPOSURE.TYPE (EXP_TYPE). The nested UseAfter is then used for a secondary lookup to determine the assigned reference.

Parameters which contain or-bars, e.g.:

```
'NRS_FIXEDSLIT|NRS_BRIGHTOBJ'
```

specify groups of values for which a file is equally applicable.

In this case the file *jwst_nirspec_pathloss_0001.fits* can be used to calibrate either NRS_FIXEDSLIT or NRS_BRIGHTOBJ.

`Or'ed` parameter combinations shown in rmaps are almost identical to the or'ed parameter combinations taken from `P\_` pattern keywords; the only difference is that rmaps do not specify the trailing or-bar required for `P\_` keyword values.

If a parameter combination maps to the value N/A, then the reference type is not applicable for that combination and CRDS returns the value N/A instead of a filename.

UseAfter Selector

The UseAfter sub-selector applies a given reference file only to datasets which occur at or after the specified date. For cases where multiple references occur prior to a dataset, CRDS chooses the most recent reference file as best.

Based on the dataset's values of:

```
META.OBSERVATION.DATE (DATE-OBS)
META.OBSERVATION.TIME (TIME-OBS)
```

CRDS will choose the appropriate reference file by comparing them to the date+time shown in the .rmap. Conceptually, the date+time shown corresponds to the value of:

```
META.REFERENCE.USEAFTER (USEAFTER)
```

from each reference file with the USEAFTER's T replaced with a space.

- In the example above, if the dataset defines:

  ```
  EXP_TYPE=NRS_MSASPEC
  DATE-OBS=1999-01-01
  TIME-OBS=00:00:00
  ```

then CRDS will select *jwst_nirspec_pathloss_0002.fits* as best.

- In the example above, if the dataset defines:

  ```
  EXP_TYPE=NRS_MSASPEC
  DATE-OBS=2001-01-01
  TIME-OBS=00:00:00
  ```

then CRDS will select *jwst_nirspec_pathloss_0007.fits* as best.

- If the dataset defines e.g.:

  ```
  DATE-OBS=1864-01-01
  ```

then no reference match exists because the observation date precedes the USEAFTER of all available reference files.

UseAfter selection is one of the rare cases where CRDS makes an apples-to-oranges match and the dataset and reference file parameters being correlated are not identical. In fact, not even the count of parameters (DATE-OBS, TIME-OBS) vs. USEAFTER is identical.

Defining Reference File Applicability

Almost all reference files supply metadata which defines how CRDS should add the file to its corresponding .rmap, i.e. each reference defines the science data parameters for which it is *initially* applicable.

When creating reference files, you will need to define a value for every CRDS matching parameter and/or define a pattern using the `P_` version of the matching parameter.

When CRDS adds a reference file to a .rmap, it uses literal matching between the value defined in the reference file and the existing values shown in the .rmap. This enables CRDS to:

1. add files to existing categories
2. replace files in existing categories
3. create new categories of files.

Because creating new categories is an unusual event which should be carefully reviewed, CRDS issues a warning when a reference file defines a new category.

Changing .rmaps to Reassign Reference Files

While reference files generally specify their intended use, sometimes different desired uses not specified in the reference file appear over time. In CRDS it is possible to alter only a .rmap to change the category or dates for which a reference file applies.

This is a fundamental CRDS feature which enables changes to reference assignment without forcing the re-delivery of an otherwise serviceable reference file. This feature is very commonly used, and the net consequence is that **.rmap categories and dates do not have to match the contents of reference files.**

It is better to view CRDS matching as a comparison between dataset parameters and a .rmap. Although references do state "initial intent", reference file metadata should not be viewed as definitive for how a file is assigned.

More Complex Matching

CRDS matching supports more complex situations than shown in the example above.

Although reference files are generally constructed so that their metadata defines the instrument modes for which they're applicable, conceptually, the values shown in .rmaps correspond to values in the dataset. Indeed, it is possible to change the values shown in the rmap so that they differ from their corresponding values in the reference file. This makes it possible to reassign reference files rather than redelivering them.

Match Parameter Combinations

For matches using combinations of multiple parameters, the Match selector keys will be shown as tuples, e.g.:

```
('NRS1|NRS2', 'ANY', 'GENERIC', '1', '1', '2048', '2048')
```

Because this match category matches either DETECTOR=NRS1 or NRS2, this single rmap entry represents two discrete parameter combinations. With multiple pattern values (not shown here), a single match category can match many different discrete combinations.

The *parkey* tuple from the NIRSPEC SUPERBIAS rmap which supplied the example match case above looks like:

```
(('META.INSTRUMENT.DETECTOR', 'META.EXPOSURE.READPATT',
'META.SUBARRAY.NAME', 'META.SUBARRAY.XSTART', 'META.SUBARRAY.YSTART',
'META.SUBARRAY.XSIZE', 'META.SUBARRAY.YSIZE'),
('META.OBSERVATION.DATE', 'META.OBSERVATION.TIME'))
```

The first sub-tuple corresponds to the Match cases, and the second sub-tuple corresponds to the nested UseAfters.

Weighted Matching

It's possible for CRDS to complete it's search without finding a unique match. To help resolve these situations, the Match algorithm uses a weighting scheme.

Each parameter with an exact match contributes a value of 1 to the weighted sum. e.g. 'NRS1' matches 'NRS1|NRS2' exactly once patterns are accounted for.

An rmap value of ANY will match any dataset value and also has a weight of 1.

An rmap value of N/A or GENERIC will match any dataset value but have a weight of 0, contributing nothing to the strength of the match.

Conceptually, the match with the highest weighting value is used. It is possible to create rmaps where ambiguity is not resolved by the weighting scheme but it works fairly well when used sparingly and isolated to as few parameters as possible.

Typically the value GENERIC corresponds to a full frame reference file which can support the calibration of any SUBARRAY by performing a cut-out.

More Information

More information about CRDS can be found in the CRDS User's Guide maintained on the CRDS server here:

https://jwst-crds.stsci.edu/static/users_guide/index.html

15.1.48 Reference Pixel Correction

Description

Class
jwst.refpix.RefPixStep

Alias
refpix

Overview

With a perfect detector and readout electronics, the signal in any given readout would differ from that in the previous readout only as a result of detected photons. In reality, the readout electronics imposes its own signal on top of this. In its simplest form, the amplifiers add a constant value to each pixel, and this constant value is different from amplifier to amplifier in a given group, and varies from group to group for a given amplifier. The magnitude of this variation is of the order of a few counts. In addition, superposed on this signal is a variation that is mainly with row number that seems to apply to all amplifiers within a group.

The `refpix` step corrects for these drifts by using the reference pixels. NIR detectors have their reference pixels in a 4-pixel wide strip around the edge of the detectors that are completely insensitive to light, while the MIR detectors have 4 columns (1 column for each amplifier) of reference pixels at the left and right edges of the detector. They also have data read through a fifth amplifier, which is called the reference output, but these data are not currently used in any refpix correction.

The effect is more pronounced for the NIR detectors than for the MIR detectors.

Input details

The input file must be a 4-D ramp and it should contain both a science (SCI) extension and a pixel data quality (PIXELDQ) extension. The PIXELDQ extension is normally populated by the `dq_init` step, so running that step is a prerequisite for the `refpix` step.

Algorithms

The algorithms for the NIR and MIR detectors are somewhat different. An entirely different algorithm for NIRSpec IRS2 readout mode is described in *IRS2*.

NIR Detector Data

1. The data from most detectors will have been rotated and/or flipped from their detector frame in order to give them the same orientation and parity in the telescope focal plane. The first step is to transform them back to the detector frame so that all NIR and MIR detectors can be treated equivalently.
2. It is assumed that a superbias correction has been performed.
3. For each integration and for each group:
    1. Calculate the mean value in the top and bottom reference pixels. The reference pixel means for each amplifier are calculated separately, and the top and bottom means are calculated separately. Optionally, the user can choose to calculate the means of odd and even columns separately by using the `--odd_even_columns` step parameter, because evidence has been found that there is a significant odd-even column effect in some datasets. Bad pixels (those whose DQ flag has the "DO_NOT_USE" bit set) are not included in the calculation of the mean.
    2. The mean is calculated as a clipped mean with a 3-sigma rejection threshold using the `scipy.stats.sigmaclip` method.
    3. Average the top and bottom reference pixel mean values
    4. Subtract each mean from all pixels that the mean is representative of, i.e. by amplifier and using the odd mean for the odd pixels and even mean for even pixels if this option is selected.
    5. If the `--use_side_ref_pixels` option is selected, use the reference pixels up the side of the A and D amplifiers to calculate a smoothed reference pixel signal as a function of row. A running median of height

set by the step parameter `side_smoothing_length` (default value 11) is calculated for the left and right side reference pixels, and the overall reference signal is obtained by averaging the left and right signals. A multiple of this signal (set by the step parameter `side_gain`, which defaults to 1.0) is subtracted from the full group on a row-by-row basis. Note that the `odd_even_rows` parameter is ignored for NIR data when the side reference pixels are processed.

6. Transform the data back to the JWST focal plane, or DMS, frame.

MIR Detector Data

1. MIR data are always in the detector frame, so no flipping/rotation is needed.
2. Subtract the first group from each group within an integration.
3. For each integration, and for each group after the first:
    1. Calculate the mean value in the reference pixels for each amplifier. The left and right side reference signals are calculated separately. Optionally, the user can choose to calculate the means of odd and even rows separately using the `--odd_even_rows` step parameter, because it has been found that there is a significant odd-even row effect. Bad pixels (those whose DQ flag has the "DO_NOT_USE" bit set) are not included in the calculation of the mean. The mean is calculated as a clipped mean with a 3-sigma rejection threshold using the `scipy.stats.sigmaclip` method. Note that the `odd_even_columns`, `use_side_ref_pixels`, `side_smoothing_length` and `side_gain` parameters are ignored for MIRI data.
    2. Average the left and right reference pixel mean values.
    3. Subtract each mean from all pixels that the mean is representative of, i.e. by amplifier and using the odd mean for the odd row pixels and even mean for even row pixels if this option is selected.
    4. Add the first group of each integration back to each group.

At the end of the refpix step, the S_REFPIX keyword is set to "COMPLETE".

NIRCam Frame 0

If a frame zero data cube is present in the input data, the image corresponding to each integration is corrected in the same way as the regular science data and passed along to subsequent pipeline steps.

Subarrays

Subarrays are treated slightly differently. Once again, the data are flipped and/or rotated to convert to the detector frame.

NIR Data

For single amplifier readout (NOUTPUTS keyword = 1):

If the `odd_even_columns` flag is set to True, then the clipped means of all reference pixels in odd-numbered columns and those in even numbered columns are calculated separately, and subtracted from their respective data columns. If the flag is False, then a single clipped mean is calculated from all of the reference pixels in each group and subtracted from each pixel.

Note: In subarray data, reference pixels are identified by the PIXELDQ array having the value of "REFERENCE_PIXEL" (defined in datamodels/dqflags.py). These values are populated when the `dq_init` step is run, so it is important to run that step before running the `refpix` step on subarray data.

Additionally, certain NIRSpec subarrays (SUB32, SUB512 and SUB512S) do not include any physical reference pixels in their readouts. For these subarrays, the first and last four image columns should not receive any incoming light with the filter+grating combinations for which they are approved for use, hence they can be used in place of actual reference pixels. The step assigns the "REFERENCE_PIXEL" DQ flag to these image columns, which then causes them to be used to perform the reference pixel correction.

If the science dataset has at least 1 group with no valid reference pixels, the step is skipped and the S_REFPIX header keyword is set to 'SKIPPED'.

The `use_side_ref_pixels`, `side_smoothing_length`, `side_gain` and `odd_even_rows` parameters are ignored for these types of data.

For 4 amplifier readout (NOUTPUTS keyword = 4):

If the NOUTPUTS keyword is 4 for a subarray exposure, then the data are calibrated the same as for full-frame exposures. The top/bottom reference values are obtained from available reference pixel regions, and the side reference values are used if available. If only 1 of the top/bottom or side reference regions are available, they are used, whereas if both are available they are averaged. If there are no top/bottom or side reference pixels available, then that part of the correction is omitted. The routine will log which parameters are valid according to whether valid reference pixels exist.

MIR Data

The refpix correction is skipped for MIRI subarray data.

NIRSpec IRS2 Readout Mode

This section describes – in a nutshell – the procedure for applying the reference pixel correction for data read out using the IRS2 readout pattern. See the JdoxIRS2 (https://jwst-docs.stsci.edu/jwst-near-infrared-spectrograph/nirspec-instrumentation/nirspec-detectors/nirspec-detector-readout-modes-and-patterns/nirspec-irs2-detector-readout-mode) page for for an overview, and see Rauscher2017 (http://adsabs.harvard.edu/abs/2017PASP..129j5003R) for details.

The raw data include both the science data and interspersed reference pixel values. The time to read out the entire detector includes not only the time to read each pixel of science ("normal") data and some of the reference pixels, but also time for the transition between reading normal data and reference pixels, as well as additional overhead at the end of each row and between frames. For example, it takes the same length of time to jump from reading normal pixels to reading reference pixels as it does to read one pixel value, about ten microseconds.

Before subtracting the reference pixel and reference output values from the science data, some processing is done on the reference values, and the CRDS reference file factors are applied. IRS2 readout is only used for full-frame data, never for subarrays. The full detector is read out by four separate amplifiers simultaneously, and the reference output is read at the same time. Each of these five readouts is the same size, 640 by 2048 pixels (for IRS2). If the CRDS reference file includes a DQ (data quality) BINTABLE extension, interleaved reference pixel values will be set to zero if they are flagged as bad in the DQ extension.

At this point the algorithm looks for intermittently bad (or suspicious) reference pixels. This is done by calculating the means and standard deviations per reference pixel column, as well as the difference between even and odd pairs; then calculates the mean and standard deviation of each of these arrays (the mean of the absolute values for the differences

array), and flag all values greater than the corresponding mean plus the standard deviation times a factor to avoid overcorrection. All suspicious pixels will be replaced by their nearest good reference pixel, or set to zero if there were no good reference pixels left (although this is unlikely to happen as there are typically only a few pixels flagged as suspicious).

The next step in this processing is to copy the science data and the reference pixel data separately to temporary 1-D arrays (both of length 712 * 2048); this is done separately for each amp output. The reference output is also copied to such an array, but there is only one of these. When copying a pixel of science or reference pixel data to a temporary array, the elements are assigned so that the array indexes increase with and correspond to the time at which the pixel value was read. That means that the change in readout direction from one amplifier to the next is taken into account when the data are copied, and that there will be gaps (array elements with zero values), corresponding to the times when reference pixels were read (or science data, depending on which is being copied), or corresponding to the overheads mentioned in the previous paragraph. The gaps will then be assigned values by interpolation (cosine-weighted, then Fourier filtered). Note that the above is done for every group.

The `alpha` and `beta` arrays that were read from the CRDS reference file are next applied, and this is done in Fourier space. These are applied to the temporary 1-D arrays of reference pixel data and to the reference output array. `alpha` and `beta` have shape (4, 712 * 2048) and data type Complex64 (stored as pairs of Float32 in the reference file). The first index corresponds to the sector number for the different output amplifiers. `alpha` is read from columns 'ALPHA_0', 'ALPHA_1', 'ALPHA_2', and 'ALPHA_3'. `beta` is read from columns 'BETA_0', 'BETA_1', 'BETA_2', and 'BETA_3'.

For each integration, the following is done in a loop over groups.

Let `k` be the output number, i.e. an index for sectors 0 through 3. Let `ft_refpix` be an array of shape (4, 712 * 2048); for each output number `k`, `ft_refpix[k]` is the Fourier transform of the temporary 1-D array of reference pixel data. Let `ft_refout` be the Fourier transform of the temporary 1-D array of reference output data. Then:

```
for k in range(4):
    ft_refpix_corr[k] = ft_refpix[k] * beta[k] + ft_refout * alpha[k]
```

For each `k`, the inverse Fourier transform of `ft_refpix_corr[k]` is the processed array of reference pixel data, which is then subtracted from the normal pixel data over the range of pixels for output `k`.

Step Arguments

The reference pixel correction step has five step-specific arguments:

- `--odd_even_columns`

If the `odd_even_columns` argument is given, the top/bottom reference signal is calculated and applied separately for even- and odd-numbered columns. The default value is True, and this argument applies to NIR data only.

- `--use_side_ref_pixels`

If the `use_side_ref_pixels` argument is given, the side reference pixels are used to calculate a reference signal for each row, which is subtracted from the data. The default value is True, and this argument applies to NIR data only.

- `--side_smoothing_length`

The `side_smoothing_length` argument is used to specify the height of the window used in calculating the running median when calculating the side reference signal. The default value is 11, and this argument applies to NIR data only when the `--use_side_ref_pixels` option is selected.

- `--side_gain`

The `side_gain` argument is used to specify the factor that the side reference signal is multiplied by before subtracting from the group row-by-row. The default value is 1.0, and this argument applies to NIR data only when the `--use_side_ref_pixels` option is selected.

- `--odd_even_rows`

If the `odd_even_rows` argument is selected, the reference signal is calculated and applied separately for even- and odd-numbered rows. The default value is True, and this argument applies to MIR data only.

- `--ovr_corr_mitigation_ftr`

This is a factor to avoid overcorrection of intermittently bad reference pixels in the IRS2 algorithm. This factor is the number of sigmas away from the mean. The default value is 3.0, and this argument applies only to NIRSpec data taken with IRS2 mode.

Reference Files

The `refpix` step uses a REFPIX reference file, but only when processing NIRSpec exposures that have been acquired using an IRS2 readout pattern. No other instruments or exposure modes require a reference file for this step.

REFPIX Reference File

REFTYPE
REFPIX

Data model
`IRS2Model` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IRS2Model.html#jwst.datamodels.IRS2Mode

The REFPIX reference file contains the complex coefficients for the correction.

Reference Selection Keywords for REFPIX

CRDS selects appropriate REFPIX references based on the following keywords. REFPIX is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
NIRSpec	INSTRUME, DETECTOR, READPATT, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for REFPIX

In addition to the standard reference file keywords listed above, the following keywords are *required* in REFPIX reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for REFPIX*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector
READPATT	model.meta.exposure.readpatt

Reference File Format

REFPIX reference files are FITS format, with two BINTABLE extensions. The FITS primary HDU does not contain a data array. The first BINTABLE extension is labeled with EXTNAME = "IRS2" and has the following column characteristics:

Column	Data type
alpha_0	float32
alpha_1	float32
alpha_2	float32
alpha_3	float32
beta_0	float32
beta_1	float32
beta_2	float32
beta_3	float32

The "alpha" arrays contain correction multipliers to the reference output, and the "beta" arrays contain correction multipliers to the interleaved reference pixels. Both arrays have 4 components, one for each sector (amplifier output). These are applied to (i.e. multiplied by) the Fourier transform of the interleaved reference pixel data. The coefficients are intrinsically complex values, but have their real and imaginary parts stored in alternating table rows, i.e. row 1 contains the real components of all coefficients and row 2 contains the corresponding imaginary components for each. This storage scheme results in a total of 2916352 (2048 * 712 * 2) rows in the table.

The second BINTABLE extension is labeled with EXTNAME = "DQ" and has the following column characteristics:

Column	Data type
output	int16
odd_even	int16
mask	uint32

This table has eight rows. The "output" column contains the amplifier output numbers: 1, 1, 2, 2, 3, 3, 4, 4. The "odd_even" column contains values 1 or 2, indicating that either the first or second pair of reference pixel reads respectively should be regarded as bad. The "mask" column contains 32-bit unsigned integer values. The interpretation of these values was described in the ESA CDP3 document as follows:

"There is also a DQ extension that holds a binary table with three columns (OUTPUT, ODD_EVEN, and MASK) and eight rows. In the current IRS2 implementation, one jumps 32 times to odd and 32 times to even reference pixels, which are then read twice consecutively. Therefore, the masks are 32 bit unsigned integers that encode bad interleaved reference pixels/columns from left to right (increasing column index) in the native detector frame. When a bit is set, the corresponding reference data should not be used for the correction."

We assume that "native detector frame" in the above description referred to the order that the data and interleaved reference pixels were read out from the detector, not the physical locations of the pixels on the detector. The difference is that the readout direction changes when going from one amplifier output to the next; that is, the pixels are read out from left to right for the first and third outputs, and they are read out from right to left for the second and fourth outputs. Furthermore, we assume that for the first amplifier output, it is the least significant bit in the value from the MASK column that corresponds to the first set of four reads of interleaved reference pixel values (reading pixels from left to right).

jwst.refpix Package

Classes

RefPixStep([name, parent, config_file, ...])	RefPixStep: Use reference pixels to correct bias drifts

RefPixStep

class jwst.refpix.**RefPixStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

RefPixStep: Use reference pixels to correct bias drifts

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'refpix'

reference_file_types = ['refpix']

spec

```
odd_even_columns = boolean(default=True) # Compute reference signal separately␣
↪for even/odd columns
use_side_ref_pixels = boolean(default=True) # Use side reference pixels for␣
↪reference signal for each row
side_smoothing_length = integer(default=11) # Median window smoothing height␣
↪for side reference signal
side_gain = float(default=1.0) # Multiplicative factor for side reference␣
↪signal before subtracting from rows
odd_even_rows = boolean(default=True) # Compute reference signal separately for␣
↪even- and odd-numbered rows
ovr_corr_mitigation_ftr = float(default=3.0) # Factor to avoid overcorrection␣
↪of bad reference pixels for IRS2
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.49 Resampling

Description

Classes
jwst.resample.ResampleStep, *jwst.resample.ResampleSpecStep*

Alias
resample, resample_spec

This routine will resample each input 2D image based on the WCS and distortion information, and will combine multiple resampled images into a single undistorted product. The distortion information should have been incorporated into the image using the *assign_wcs* step.

The `resample` step can take as input either:

1. a single 2D input image
2. an association table (in json format)

The defined parameters for the drizzle operation itself get provided by the DRIZPARS reference file (from CRDS). The exact values used depends on the number of input images being combined and the filter being used. Other information may be added as selection criteria later, but for now, only basic information is used.

The output product gets defined using the WCS information of all inputs, even if it is just a single input image. The output WCS defines a field-of-view that encompasses the undistorted footprints on the sky of all the input images with the same orientation and plate scale as the first listed input image.

This step uses the interface to the C-based cdriz routine to do the resampling via the drizzle method. The input-to-output pixel mapping is determined via a mapping function derived from the WCS of each input image and the WCS of the defined output product. This mapping function gets passed to cdriz to drive the actual drizzling to create the output product.

Context Image

In addition to image data, resample step also creates a "context image" stored in the `con` attribute in the output data model or `'CON'` extension of the FITS file. Each pixel in the context image is a bit field that encodes information about which input image has contributed to the corresponding pixel in the resampled data array. Context image uses 32 bit integers to encode this information and hence it can keep track of only 32 input images. First bit corresponds to the first input image, second bit corrsponds to the second input image, and so on. If the number of input images is larger than 32, then it is necessary to have multiple context images ("planes") to hold information about all input images with the first plane encoding which of the first 32 images contributed to the output data pixel, second plane representing next 32 input images (number 33-64), etc. For this reason, context array is a 3D array of the type `numpy.int32` (https://numpy.org/devdocs/reference/arrays.scalars.html#numpy.int32) and shape `(np, ny, nx)` where `nx` and `ny`

are dimensions of image's data. `np` is the number of "planes" equal to `(number of input images - 1) // 32 + 1`. If a bit at position `k` in a pixel with coordinates `(p, y, x)` is 0 then input image number `32 * p + k` (0-indexed) did not contribute to the output data pixel with array coordinates `(y, x)` and if that bit is 1 then input image number `32 * p + k` did contribute to the pixel `(y, x)` in the resampled image.

As an example, let's assume we have 8 input images. Then, when `'CON'` pixel values are displayed using binary representation (and decimal in parenthesis), one could see values like this:

```
00000001 (1) - only first input image contributed to this output pixel;
00000010 (2) - 2nd input image contributed;
00000100 (4) - 3rd input image contributed;
10000000 (128) - 8th input image contributed;
10000100 (132=128+4) - 3rd and 8th input images contributed;
11001101 (205=1+4+8+64+128) - input images 1, 3, 4, 7, 8 have contributed
to this output pixel.
```

In order to test if a specific input image contributed to an output pixel, one needs to use bitwise operations. Using the example above, to test whether input images number 4 and 5 have contributed to the output pixel whose corresponding `'CON'` value is 205 (11001101 in binary form) we can do the following:

```
>>> bool(205 & (1 << (5 - 1)))  # (205 & 16) = 0 (== 0 => False): did NOT contribute
False
>>> bool(205 & (1 << (4 - 1)))  # (205 & 8) = 8 (!= 0 => True): did contribute
True
```

In general, to get a list of all input images that have contributed to an output resampled pixel with image coordinates `(x, y)`, and given a context array `con`, one can do something like this:

```
>>> import numpy as np
>>> np.flatnonzero([v & (1 << k) for v in con[:, y, x] for k in range(32)])
```

For convenience, this functionality was implemented in the *decode_context()* function.

Spectroscopic Data

Use the `resample_spec` step for spectroscopic data. The dispersion direction is needed for this case, and this is obtained from the DISPAXIS keyword. For the NIRSpec Fixed Slit mode, the `resample_spec` step will be skipped if the input is a rateints product, as 3D input for the mode is not supported.

References

A full description of the drizzling algorithm can be found in Fruchter and Hook, PASP 2002 (https://doi.org/10.1086/338393). A description of the inverse variance map method can be found in Casertano et al., AJ 2000 (https://doi.org/10.1086/316851), see Appendix A2. A description of the drizzle parameters and other useful drizzle-related resources can be found at DrizzlePac Handbook (http://drizzlepac.stsci.edu).

Step Arguments

The `resample` step has the following optional arguments that control the behavior of the processing and the characteristics of the resampled image.

`--pixfrac` (float, default=1.0)
The fraction by which input pixels are "shrunk" before being drizzled onto the output image grid, given as a real number between 0 and 1.

`--kernel` (str, default='square')
The form of the kernel function used to distribute flux onto the output image. Available kernels are `square`, `gaussian`, `point`, `tophat`, `turbo`, `lanczos2`, and `lanczos3`.

`--pixel_scale_ratio` (float, default=1.0)
Ratio of input to output pixel scale. A value of 0.5 means the output image would have 4 pixels sampling each input pixel. Ignored when `pixel_scale` or `output_wcs` are provided.

`--pixel_scale` (float, default=None)
Absolute pixel scale in `arcsec`. When provided, overrides `pixel_scale_ratio`. Ignored when `output_wcs` is provided.

`--rotation` (float, default=None)
Position angle of output image's Y-axis relative to North. A value of 0.0 would orient the final output image to be North up. The default of `None` (https://docs.python.org/3/library/constants.html#None) specifies that the images will not be rotated, but will instead be resampled in the default orientation for the camera with the x and y axes of the resampled image corresponding approximately to the detector axes. Ignored when `pixel_scale` or `output_wcs` are provided.

`--crpix` (tuple of float, default=None)
Position of the reference pixel in the image array in the `x, y` order. If `crpix` is not specified, it will be set to the center of the bounding box of the returned WCS object. When supplied from command line, it should be a comma-separated list of floats. Ignored when `output_wcs` is provided.

`--crval` (tuple of float, default=None)
Right ascension and declination of the reference pixel. Automatically computed if not provided. When supplied from command line, it should be a comma-separated list of floats. Ignored when `output_wcs` is provided.

`--output_shape` (tuple of int, default=None)
Shape of the image (data array) using "standard" `nx` first and `ny` second (as opposite to the `numpy.ndarray` convention - `ny` first and `nx` second). This value will be assigned to `pixel_shape` and `array_shape` properties of the returned WCS object. When supplied from command line, it should be a comma-separated list of integers `nx, ny`.

Note: Specifying `output_shape` *is required* when the WCS in `output_wcs` does not have `bounding_box` property set.

`--output_wcs` (str, default='')
File name of a `ASDF` file with a GWCS stored under the `"wcs"` key under the root of the file. The output image size is determined from the bounding box of the WCS (if any). Argument `output_shape` overrides computed image size and it is required when output WCS does not have `bounding_box` property set or if `pixel_shape` or `array_shape` keys (see below) are not provided.

Additional information may be stored under other keys under the root of the file. Currently, the following keys are recognized:

- `pixel_area`: Indicates average pixel area of the output WCS in units of steradians. When provided, this value will be used for updating photometric quantities `PIXAR_SR` and `PIXAR_A2` of the output image. If `pixel_area` is not provided, the code will attempt to estimate this value from the WCS.

- `pixel_shape`: dimensions of the output image in the order (nx, ny). Overrides the value of `array_shape` if provided.
- `array_shape`: shape of the output image in `numpy` order: (ny, nx).

Note: When `output_wcs` is specified, WCS-related arguments such as `pixel_scale_ratio`, `pixel_scale`, `rotation`, `crpix`, and `crval` will be ignored.

`--fillval` (str, default='INDEF')
The value to assign to output pixels that have zero weight or do not receive any flux from any input pixels during drizzling.

`--weight_type` (str, default='ivm')
The weighting type for each input image. If `weight_type=ivm` (the default), the scaling value will be determined per-pixel using the inverse of the read noise (VAR_RNOISE) array stored in each input image. If the VAR_RNOISE array does not exist, the variance is set to 1 for all pixels (equal weighting). If `weight_type=exptime`, the scaling value will be set equal to the exposure time found in the image header.

`--single` (bool, default=False)
If set to `True` (https://docs.python.org/3/library/constants.html#True), resample each input image into a separate output. If `False` (https://docs.python.org/3/library/constants.html#False) (the default), each input is resampled additively (with weights) to a common output

`--blendheaders` (bool, default=True)
Blend metadata from all input images into the resampled output image.

`--allowed_memory` (float, default=None)
Specifies the fractional amount of free memory to allow when creating the resampled image. If `None`, the environment variable `DMODEL_ALLOWED_MEMORY` is used. If not defined, no check is made. If the resampled image would be larger than specified, an `OutputTooLargeError` exception will be generated.

For example, if set to `0.5`, only resampled images that use less than half the available memory can be created.

`--in_memory` (boolean, default=True)
Specifies whether or not to load and create all images that are used during processing into memory. If `False`, input files are loaded from disk when needed and all intermediate files are stored on disk, rather than in memory.

Reference File

The `resample` step uses the DRIZPARS reference file.

DRIZPARS Reference File

REFTYPE
DRIZPARS

Data model
`DrizParsModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.DrizParsModel.html#jwst.datamodels.D

The DRIZPARS reference file contains various drizzle parameter values that control the characteristics of a drizzled image and how it is built.

Reference Selection Keywords for DRIZPARS

CRDS selects appropriate DRIZPARS references based on the following keywords. DRIZPARS is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME
NIRCam	INSTRUME
NIRISS	INSTRUME

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for DRIZPARS

No additional specific keywords are required in DRIZPARS reference files, because CRDS selection is based only on the instrument name (see *Reference Selection Keywords for DRIZPARS*).

Reference File Format

DRIZPARS reference files are FITS format, with 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
DRIZPARS	BINTABLE	2	TFIELDS = 7	N/A

The DRIZPARS extension contains various step parameter values to be used when processing certain types of image collections. The first two columns (numimages and filter) are used as row selectors within the table. Image collections

that match those selectors then use the parameter values specified in the remainder of that table row. The table contains the following 7 columns:

TTYPE	TFORM	Description
numimages	integer	The number of images to be combined
filter	integer	The filter used to obtain the images
pixfrac	float	The pixel "shrinkage" fraction
kernel	string	The kernel function used to distribute flux
fillval	float	Value assigned to pixels with no input flux
wht_type	sting	The input image weighting type
stepsize	integer	Output WCS grid interpolation step size

Python Step Interface: ResampleStep()

jwst.resample.resample_step Module

Classes

ResampleStep([name, parent, config_file, ...])	Resample input data onto a regular grid using the drizzle algorithm.

ResampleStep

class jwst.resample.resample_step.**ResampleStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Resample input data onto a regular grid using the drizzle algorithm.

Note: When supplied via `output_wcs`, a custom WCS overrides other custom WCS parameters such as `output_shape` (now computed from by `output_wcs.bounding_box`), `crpix`

Parameters

input (*JwstDataModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel.html#jwst.d *or* `Association`) – Single filename for either a single image or an association table.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.

- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

get_drizpars(ref_filename, input_models)	Extract drizzle parameters from reference file.
process(input)	This is where real work happens.
update_fits_wcs(model)	Update FITS WCS keywords of the resampled image.

Attributes Documentation

class_alias = 'resample'

reference_file_types = ['drizpars']

spec

```
pixfrac = float(default=1.0) # change back to None when drizpar reference files␣
↪are updated
kernel = string(default='square') # change back to None when drizpar reference␣
↪files are updated
fillval = string(default='INDEF' ) # change back to None when drizpar reference␣
↪files are updated
weight_type = option('ivm', 'exptime', None, default='ivm')  # change back to␣
↪None when drizpar ref update
output_shape = int_list(min=2, max=2, default=None)  # [x, y] order
crpix = float_list(min=2, max=2, default=None)
crval = float_list(min=2, max=2, default=None)
rotation = float(default=None)
pixel_scale_ratio = float(default=1.0) # Ratio of input to output pixel scale
pixel_scale = float(default=None) # Absolute pixel scale in arcsec
output_wcs = string(default='')  # Custom output WCS.
single = boolean(default=False)
blendheaders = boolean(default=True)
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image.
in_memory = boolean(default=True)
```

Methods Documentation

get_drizpars(*ref_filename*, *input_models*)

Extract drizzle parameters from reference file.

This method extracts parameters from the drizpars reference file and uses those to set defaults on the following ResampleStep configuration parameters:

pixfrac = float(default=None) kernel = string(default=None) fillval = string(default=None) wht_type = option('ivm', 'exptime', None, default=None)

Once the defaults are set from the reference file, if the user has used a resample.cfg file or run ResampleStep using command line args, then these will overwrite the defaults pulled from the reference file.

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

update_fits_wcs(*model*)

Update FITS WCS keywords of the resampled image.

Class Inheritance Diagram

Python Interface to Drizzle: ResampleData()

jwst.resample.resample Module

Classes

OutputTooLargeError	Raised when the output is too large for in-memory instantiation
ResampleData(input_models[, output, single, ...])	This is the controlling routine for the resampling process.

OutputTooLargeError

exception jwst.resample.resample.**OutputTooLargeError**

Raised when the output is too large for in-memory instantiation

ResampleData

class jwst.resample.resample.**ResampleData**(*input_models*, *output=None*, *single=False*, *blendheaders=True*, *pixfrac=1.0*, *kernel='square'*, *fillval='INDEF'*, *wht_type='ivm'*, *good_bits=0*, *pscale_ratio=1.0*, *pscale=None*, ***kwargs*)

Bases: `object` (https://docs.python.org/3/library/functions.html#object)

This is the controlling routine for the resampling process.

Notes

This routine performs the following operations:

```
1. Extracts parameter settings from input model, such as pixfrac,
   weight type, exposure time (if relevant), and kernel, and merges
   them with any user-provided values.
2. Creates output WCS based on input images and define mapping function
   between all input arrays and the output array.
3. Updates output data model with output arrays from drizzle, including
   a record of metadata from all input models.
```

Parameters

- **input_models** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of objects*) – list of data models, one for each input image
- **output** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – filename for output
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Other parameters.

Note: `output_shape` is in the `x, y` order.

Note: `in_memory` controls whether or not the resampled array from `resample_many_to_many()` should be kept in memory or written out to disk and deleted from memory. Default value is `True` (https://docs.python.org/3/library/constants.html#True) to keep all products in memory.

Methods Summary

blend_output_metadata(output_model)	Create new output metadata based on blending all input metadata.
do_drizzle()	Pick the correct drizzling mode based on self.single
drizzle_arrays(insci, inwht, input_wcs, ...)	Low level routine for performing 'drizzle' operation on one image.
resample_many_to_many()	Resample many inputs to many outputs where outputs have a common frame.
resample_many_to_one()	Resample and coadd many inputs to a single output.
resample_variance_array(name, output_model, ...)	Resample variance arrays from self.input_models to the output_model
update_exposure_times(output_model)	Modify exposure time metadata in-place

Methods Documentation

blend_output_metadata(*output_model*)

Create new output metadata based on blending all input metadata.

do_drizzle()

Pick the correct drizzling mode based on self.single

static drizzle_arrays(*insci*, *inwht*, *input_wcs*, *output_wcs*, *outsci*, *outwht*, *outcon*, *uniqid=1*, *xmin=0*, *xmax=0*, *ymin=0*, *ymax=0*, *iscale=1.0*, *pixfrac=1.0*, *kernel='square'*, *fillval='INDEF'*, *wtscale=1.0*)

Low level routine for performing 'drizzle' operation on one image.

The interface is compatible with STScI code. All images are Python ndarrays, instead of filenames. File handling (input and output) is performed by the calling routine.

Parameters

- **insci** (*2d array*) – A 2d numpy array containing the input image to be drizzled.
- **inwht** (*2d array*) – A 2d numpy array containing the pixel by pixel weighting. Must have the same dimensions as insci. If none is supplied, the weighting is set to one.
- **input_wcs** (*gwcs.WCS object*) – The world coordinate system of the input image.
- **output_wcs** (*gwcs.WCS object*) – The world coordinate system of the output image.
- **outsci** (*2d array*) – A 2d numpy array containing the output image produced by drizzling. On the first call it should be set to zero. Subsequent calls it will hold the intermediate results. This is modified in-place.
- **outwht** (*2d array*) – A 2d numpy array containing the output counts. On the first call it should be set to zero. On subsequent calls it will hold the intermediate results. This is modified in-place.
- **outcon** (*2d or 3d array, optional*) – A 2d or 3d numpy array holding a bitmap of which image was an input for each output pixel. Should be integer zero on first call. Subsequent calls hold intermediate results. This is modified in-place.
- **uniqid** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – The id number of the input image. Should be one the first time this function is called and incremented by one on each subsequent call.

- **xmin** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – This and the following three parameters set a bounding rectangle on the input image. Only pixels on the input image inside this rectangle will have their flux added to the output image. Xmin sets the minimum value of the x dimension. The x dimension is the dimension that varies quickest on the image. All four parameters are zero based, counting starts at zero.
- **xmax** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – Sets the maximum value of the x dimension on the bounding box of the input image. If `xmax = 0`, no maximum will be set in the x dimension (all pixels in a row of the input image will be resampled).
- **ymin** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – Sets the minimum value in the y dimension on the bounding box. The y dimension varies less rapidly than the x and represents the line index on the input image.
- **ymax** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – Sets the maximum value in the y dimension. If `ymax = 0`, no maximum will be set in the y dimension (all pixels in a column of the input image will be resampled).
- **iscale** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – A scale factor to be applied to pixel intensities of the input image before resampling.
- **pixfrac** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The fraction of a pixel that the pixel flux is confined to. The default value of 1 has the pixel flux evenly spread across the image. A value of 0.5 confines it to half a pixel in the linear dimension, so the flux is confined to a quarter of the pixel area when the square kernel is used.
- **kernel** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the kernel used to combine the input. The choice of kernel controls the distribution of flux over the kernel. The kernel names are: "square", "gaussian", "point", "tophat", "turbo", "lanczos2", and "lanczos3". The square kernel is the default.
- **fillval** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The value a pixel is set to in the output if the input image does not overlap it. The default value of INDEF does not set a value.

Returns

- **A tuple with three values** (*a version string, the number of pixels*)
- *on the input image that do not overlap the output image, and the*
- *number of complete lines on the input image that do not overlap the*
- *output input image.*

`resample_many_to_many()`

Resample many inputs to many outputs where outputs have a common frame.

Coadd only different detectors of the same exposure, i.e. map NRCA5 and NRCB5 onto the same output image, as they image different areas of the sky.

Used for outlier detection

`resample_many_to_one()`

Resample and coadd many inputs to a single output.

Used for stage 3 resampling

resample_variance_array(*name*, *output_model*, *iscale*)

Resample variance arrays from self.input_models to the output_model

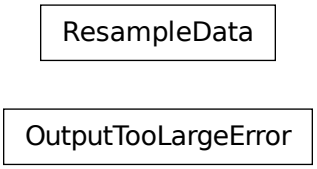
Resample the `name` variance array to the same name in output_model, using a cumulative sum.

This modifies output_model in-place.

update_exposure_times(*output_model*)

Modify exposure time metadata in-place

Class Inheritance Diagram



Resample Utilities

jwst.resample.resample_utils Module

Functions

decode_context(context, x, y)	Get 0-based indices of input images that contributed to (resampled) output pixel with coordinates `x` and `y`.

decode_context

jwst.resample.resample_utils.**decode_context**(*context*, *x*, *y*)

Get 0-based indices of input images that contributed to (resampled) output pixel with coordinates `x` and `y`.

Parameters

- **context** (*numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray)) – A 3D ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) of integral data type.
- **x** (*int* (https://docs.python.org/3/library/functions.html#int), *list* (https://docs.python.org/3/library/stdtypes.html#list) *of integers*, *numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) *of integers*) – X-coordinate of pixels to decode (3rd index into the `context` array)

- **y** (*int* (https://docs.python.org/3/library/functions.html#int), *list* (https://docs.python.org/3/library/stdtypes.html#list) *of integers*, *numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) *of integers*) – Y-coordinate of pixels to decode (2nd index into the `context` array)

Returns

- A list of `numpy.ndarray` (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) objects each containing indices of input images
- that have contributed to an output pixel with coordinates `x` and `y`.
- *The length of returned list is equal to the number of input coordinate*
- arrays `x` and `y`.

Examples

An example context array for an output image of array shape `(5, 6)` obtained by resampling 80 input images.

```
>>> import numpy as np
>>> from jwst.resample.resample_utils import decode_context
>>> con = np.array(
...     [[[0, 0, 0, 0, 0, 0],
...       [0, 0, 0, 36196864, 0, 0],
...       [0, 0, 0, 0, 0, 0],
...       [0, 0, 0, 0, 0, 0],
...       [0, 0, 537920000, 0, 0, 0]],
...      [[0, 0, 0, 0, 0, 0,],
...       [0, 0, 0, 67125536, 0, 0],
...       [0, 0, 0, 0, 0, 0],
...       [0, 0, 0, 0, 0, 0],
...       [0, 0, 163856, 0, 0, 0]],
...      [[0, 0, 0, 0, 0, 0],
...       [0, 0, 0, 8203, 0, 0],
...       [0, 0, 0, 0, 0, 0],
...       [0, 0, 0, 0, 0, 0],
...       [0, 0, 32865, 0, 0, 0]]],
...     dtype=np.int32
... )
>>> decode_context(con, [3, 2], [1, 4])
[array([ 9, 12, 14, 19, 21, 25, 37, 40, 46, 58, 64, 65, 67, 77]),
 array([ 9, 20, 29, 36, 47, 49, 64, 69, 70, 79])]
```

jwst.resample Package

Classes

ResampleStep([name, parent, config_file, ...])	Resample input data onto a regular grid using the drizzle algorithm.
ResampleSpecStep([name, parent, ...])	ResampleSpecStep: Resample input data onto a regular grid using the drizzle algorithm.

ResampleStep

class jwst.resample.**ResampleStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

Resample input data onto a regular grid using the drizzle algorithm.

Note: When supplied via output_wcs, a custom WCS overrides other custom WCS parameters such as output_shape (now computed from by output_wcs.bounding_box), crpix

Parameters

input (*JwstDataModel* (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel.html#jwst.d *or* Association) – Single filename for either a single image or an association table.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

get_drizpars(ref_filename, input_models)	Extract drizzle parameters from reference file.
process(input)	This is where real work happens.
update_fits_wcs(model)	Update FITS WCS keywords of the resampled image.

Attributes Documentation

class_alias = 'resample'

reference_file_types = ['drizpars']

spec

```
pixfrac = float(default=1.0) # change back to None when drizpar reference files␣
↪are updated
kernel = string(default='square') # change back to None when drizpar reference␣
↪files are updated
fillval = string(default='INDEF' ) # change back to None when drizpar reference␣
↪files are updated
weight_type = option('ivm', 'exptime', None, default='ivm')  # change back to␣
↪None when drizpar ref update
output_shape = int_list(min=2, max=2, default=None)  # [x, y] order
crpix = float_list(min=2, max=2, default=None)
crval = float_list(min=2, max=2, default=None)
rotation = float(default=None)
pixel_scale_ratio = float(default=1.0) # Ratio of input to output pixel scale
pixel_scale = float(default=None) # Absolute pixel scale in arcsec
output_wcs = string(default='')  # Custom output WCS.
single = boolean(default=False)
blendheaders = boolean(default=True)
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image.
in_memory = boolean(default=True)
```

Methods Documentation

get_drizpars(*ref_filename*, *input_models*)

Extract drizzle parameters from reference file.

This method extracts parameters from the drizpars reference file and uses those to set defaults on the following ResampleStep configuration parameters:

pixfrac = float(default=None) kernel = string(default=None) fillval = string(default=None) wht_type = option('ivm', 'exptime', None, default=None)

Once the defaults are set from the reference file, if the user has used a resample.cfg file or run ResampleStep using command line args, then these will overwrite the defaults pulled from the reference file.

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

update_fits_wcs(*model*)

Update FITS WCS keywords of the resampled image.

ResampleSpecStep

class jwst.resample.**ResampleSpecStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *ResampleStep*

ResampleSpecStep: Resample input data onto a regular grid using the drizzle algorithm.

Parameters

input (MultiSlitModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.
ModelContainer, Association) – A singe datamodel, a container of datamodels, or an association file

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias

Methods Summary

process(input)	This is where real work happens.
update_slit_metadata(model)	Update slit attributes in the resampled slit image.

Attributes Documentation

class_alias = 'resample_spec'

Methods Documentation

process(*input*)

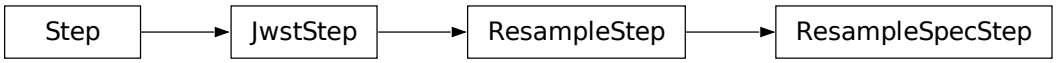
This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

update_slit_metadata(*model*)

Update slit attributes in the resampled slit image.

This is needed because model.slit attributes are not in model.meta, so the normal update() method doesn't work with them. Updates output_model in-place.

Class Inheritance Diagram



15.1.50 Reset Correction

Description

Class

jwst.reset.ResetStep

Alias

reset

The reset correction is a MIRI step that attempts to correct for the reset anomaly effect. This effect is caused by the non-ideal behavior of the FET upon resetting in the dark causing the initial frames in an integration to be offset from their expected values. Another MIRI effect caused by resetting the detectors is the RSCD effect (see *rscd*).

Assumptions

The reset correction is a MIRI-specific correction. It will not be applied to data from other instruments.

Background

For MIRI exposures, the initial groups in each integration suffer from two effects related to the resetting of the detectors. The first effect is that the first few groups after a reset do not fall on the expected linear accumulation of signal. The most significant deviations occur in groups 1 and 2. This behavior is relatively uniform detector-wide. The second effect, on the other hand, is the appearance of significant extra spatial structure in these initial groups, before fading out in later groups.

The reset anomaly effect fades out by ~group 15 for full array data. It takes a few more groups for the effect to fade away on subarray data. The time constant of the effect seems to be closely related to the group number and not time since reset.

For multiple integration data, the reset anomaly also varies in amplitude for the first few integrations before settling down to a relatively constant correction for integrations greater than the second integration for full array data. Because of the shorter readout time, the subarray data requires a few more integrations before the effect is relatively stable from integration to integration.

Algorithm

The reset correction step applies the reset reference file. The reset reference file contains an integration dependent correction for the first N groups, where N is defined by the reset correction reference file.

The format of the reset reference file is NCols X NRows X NGroups X NInts. For full frame data, the current implementation uses a reset anomaly reference file, which contains a correction for the first 15 groups for all integrations. The reference file contains two corrections: one for the first integration and a second one for all other integrations. The correction was determined so that the correction is forced to be zero on group 15. For each integration in the input science data, the reset corrections are subtracted, group-by-group, integration-by- integration. If the input science data contains more groups than the reset correction, then correction for those groups is zero. If the input science data contains more integrations than the reset correction then the correction corresponding to the last integration in the reset file is used.

There is a single, NCols X NRowss, DQ flag image for all the integrations. The reset DQ flag array are combined with the science PIXELDQ array using numpy's bitwise_or function. The ERR arrays of the science data are currently not modified at all.

Subarrays

The reset correction is subarray-dependent, therefore this step makes no attempt to extract subarrays from the reset reference file to match input subarrays. It instead relies on the presence of matching subarray reset reference files in the CRDS. In addition, the number of NGROUPS and NINTS for subarray data varies from the full array data as well as from each other.

Reference File Types

The `reset` correction step uses a RESET reference file.

RESET Reference File

REFTYPE
RESET

Data model
`ResetModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ResetModel.html#jwst.datamodels.ResetMo

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Reference File Format

The reset reference files are FITS files with 3 IMAGE extensions and 1 BINTABLE extension. The FITS primary data array is assumed to be empty. The characteristics of the three image extension are as follows:

EXTNAME	NAXIS	Dimensions	Data type
SCI	4	ncols x nrows x ngroups x nint	float
ERR	4	ncols x nrows x ngroups x nint	float
DQ	2	ncols x nrows	integer

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

The SCI and ERR data arrays are 4-D, with dimensions of ncols x nrows x ngroups X nints, where ncols x nrows matches the dimensions of the raw detector readout mode for which the reset applies. The reference file contains the number of NGroups planes required for the correction to be zero on the last plane Ngroups plane. The correction for the first few integrations varies and eventually settles down to a constant correction independent of integration number.

Step Arguments

The reset correction has no step-specific arguments.

jwst.reset Package

Classes

ResetStep([name, parent, config_file, ...])	ResetStep: Performs a reset correction by subtracting the reset correction reference data from the input science data model.

ResetStep

class jwst.reset.**ResetStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

ResetStep: Performs a reset correction by subtracting the reset correction reference data from the input science data model.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'reset'

reference_file_types = ['reset']

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.51 Residual Fringe

Description

Class
`jwst.residual_fringe.ResidualFringeStep`

Alias
residual_fringe

The JWST pipeline contains two steps devoted to the removal of fringes on MIRI MRS images. The first correction is applied in the `fringe_step` in the *calwebb_spec2* pipeline and consists in dividing detector-level data by a fringe-flat and is described in the *fringe* step. Applying the fringe flat should eliminate fringes from spectra of spatially extended sources, however residual fringes can remain. For spatially unresolved (point) sources or extended sources with structure, applying the fringe flat will undoubtedly leave residual fringes since these produce different fringe patterns on the detector than accounted for by the fringe flat. The second step for fringe removal is the `residual_fringe_step`. This step is part of the *calwebb_spec2* pipeline, but currently it is skipped by default. To apply this step set the step parameter, `--skip = False`. This step is applied after *photom*, but before *cube_build*.

The `residual_fringe` step can accept several different forms of input data, including:

1. a single file containing a 2-D IFU image

2. a data model (`IFUImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.IFUImageModel.html#jwst.data containing a 2-D IFU image
3. an association table (in json format) containing a single input file

Assumptions

This step only works on MIRI MRS data. It is assumed that the calwebb_spec2 pipeline has been run on the data. In addition, the detection of residual fringes are better determined if the `mrs_imatch` step has also been applied to the data.

Fringe Background Information

As is typical for spectrometers, the MIRI MRS detectors are affected by fringes. These are periodic gain modulations caused by standing waves between parallel surfaces in the optical path, acting as a slow-finesse Fabry-Pérot etalons. In the MRS, the principal fringe sources are the detector layers. A detailed detailed discussion on these fringe components can be found in Argyriou, I., Wells, M., Glasse, A., et al. 2020, A&A, 641, A150 and Wells, M., Pel, J.-W., Glasse, A., et al. 2015, PASP, 127, 646.

The primary MRS fringe, observed in all MRS bands, is caused by the etalons between the anti-reflection coating and lower layers, encompassing the detector substrate and the infrared-active layer. Since the thickness of the substrate is not the same in the SW and LW detectors, the fringe frequency will differ in the two detectors. Up to 16 microns, this fringe is produced by the anti-reflection coating and pixel metalization etalons, whereas above 16 microns it is produced by the anti-reflection coating and bottom contact etalon, resulting in a different fringe frequency. The information in the fringe frequency reference file is used to determine, for each MRS band, the frequencies to fit to this main fringe component. The residual fringes are corrected for by fitting and removing sinusoidal gain to the detector level data.

Step Arguments

The residual fringe step has two step arguments that can be used to specify wavelength regions in which no correction will be determined. The two arguments give lists of minimum and maximum wavelength values, respectively, for the regions to be ignored. The two lists must contain an equal number of elements.

`ignore_region_min` [float, default = None]
The minimum wavelengths for the region(s) to be ignored, given as a comma-separated list.

`ignore_region_max` [float, default = None]
The maximum wavelengths for the region(s) to be ignored, given as a comma-separated list.

Reference Files

The `residual_fringe` step uses the FRINGEFREQ reference file.

FRINGEFREQ reference file

REFTYPE
FRINGEFREQ

Data models
`FringeFreqModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.FringeFreqModel.html#jwst.datamod

The FRINGEFREQ reference files contain parameter values used to correct MIRI MRS images for residual fringes that remain after applying the fringe flat.

Reference Selection Keywords for FRINGEFREQ

CRDS selects appropriate FRINGEFREQ reference file based on the following keywords. FRINGEFREQ is not applicable for instruments not in the table.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DETECTOR, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for FRINGEFREQ

In addition to the standard reference file keywords listed above, the following keywords are *required* in FRINGEFREQ reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for FRINGEFREQ*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector
EXP_TYPE	model.meta.exposure.type

Reference File Format

The FRINGEFREQ reference files are FITS format, with 4 BINTABLE extensions. The FITS primary data array is assumed to be empty. The format and content of the FRINGEFREQ reference file is

EXTNAME	XTENSION	Dimensions
RFC_FREQ_SHORT	BINTABLE	TFIELDS = 7
RFC_FREQ_MEDIUM	BINTABLE	TFIELDS = 7
RFC_FREQ_LONG	BINTABLE	TFIELDS = 7
MAX_AMP	BINTABLE	TFIELDS = 2

The formats of the individual table extensions are listed below.

Table	Column	Data type	Units
RFC_FREQ_SHORT	SLICE	double	N/A
	FFREQ	double	cm^{-1}
	DFFREQ	double	cm^{-1}
	MIN_NFRIGES	int	N/A
	MAX_NFRINGES	int	N/A
	MIN_SNR	double	cm^{-1}
	PGRAM_RES	double	cm^{-1}
FRC_FREQ_MEDIUM	SLICE	double	N/A
	FFREQ	double	cm^{-1}
	DFFREQ	double	cm^{-1}
	MIN_NFRIGES	int	N/A
	MAX_NFRINGES	int	N/A
	MIN_SNR	double	cm^{-1}
	PGRAM_RES	double	cm^{-1}
RFC_FREQ_LONG	SLICE	double	N/A
	FFREQ	double	cm^{-1}
	DFFREQ	double	cm^{-1}
	MIN_NFRIGES	int	N/A
	MAX_NFRINGES	int	N/A
	MIN_SNR	double	cm^{-1}
	PGRAM_RES	double	cm^{-1}
MAX_AMP	WAVELENGTH	double	micron
	AMPLITUDE	double	N/A

These reference files contain tables for each MIRI band giving the fringe frequencies and other parameters for each band to fit and remove residual fringes.

The reference table descriptions:

- **RFC_FREQ_SHORT** table contains the fringe frequencies and parameters for the SHORT band.
- **RFC_FREQ_MEDIUM** table contains the fringe frequencies and parameters for the MEDIUM band.
- **RFC_FREQ_LONG** table contains the fringe frequencies and parameters for the LONG band.
- **MAX_AMP** table contains a wavelength dependent maximum amplitude which is use for feature identification and fit rejection.

jwst.residual_fringe.residual_fringe_step Module

Classes

ResidualFringeStep([name, parent, ...])	ResidualFringeStep: Apply residual fringe correction to a science image using parameters in the residual fringe reference file.

ResidualFringeStep

class jwst.residual_fringe.residual_fringe_step.**ResidualFringeStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

ResidualFringeStep: Apply residual fringe correction to a science image using parameters in the residual fringe reference file.

Parameters

input_data (*asn file or single file*) –

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'residual_fringe'

reference_file_types = ['fringefreq', 'regions']

spec

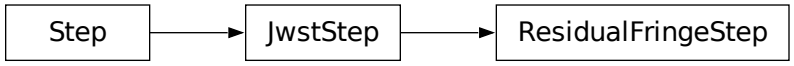
```
skip = boolean(default=True)
save_intermediate_results  = boolean(default = False)
search_output_file = boolean(default = False)
ignore_region_min = list(default = None)
ignore_region_max = list(default = None)
suffix = string(default = 'residual_fringe')
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.52 Reset Switch Charge Decay (RSCD) Correction

Description

Class
jwst.rscd.RscdStep

Alias
rscd

Assumptions

This correction is currently only implemented for MIRI data and is only applied to integrations after the first integration (i.e. this step does not correct the first integration). It is assumed this step occurs before the dark subtraction, but after linearity correction.

Background

The MIRI Focal Plane System (FPS) consists of the detectors and the electronics to control them. There are a number of non-ideal detector and readout effects that produce reset offsets, nonlinearities at the start of an integration, non-linear ramps with increasing signal, latent images, and drifts in the slopes.

The manner in which the MIRI readout electronics operate have been shown to be the source of the ramp offsets, nonlinearities at the start of the integration, and overall changes in slopes. Basically the MIRI reset electronics use field effect transistors (FETs) in their operation. The FET acts as a switch to allow charge to build up and to also initialize (clear) the charge. However, the reset FETS do not instantaneously reset the level, instead the exponential adjustment of the FET after a reset causes the initial frames in an integration to be offset from their expected values. Between exposures the MIRI detectors are continually reset; however for a multiple integration exposure there is a single reset between integrations. The effects of this decay are not measurable in the first integration because a number of resets have occurred from the last exposure and the effect has decayed away by the time it takes to read out the last exposure, set up the next exposure and begin exposing. There are low level reset effects in the first integration that are related to the strength of the dark current and can be removed with an integration-dependent dark.

The Reset Switch Charge Decay (RSCD) step corrects for these effects by simply flagging the first N groups as DO_NOT_USE. An actual correction algorithm allowing for the first N groups to be used is under development.

Algorithm

This correction is only applied to integrations > 1. This step flags the N groups at the beginning of all 2nd and higher integrations as bad (the "DO_NOT_USE" bit is set in the GROUPDQ flag array), but only if the total number of groups in each integration is greater than N+3. This results in the data contained in the the first N groups being excluded from subsequent steps, such as *jump detection* and *ramp_fitting*. No flags are added if NGROUPS <= N+3, because doing so would leave too few good groups to work with in later steps.

Only the GROUPDQ array is modified. The SCI, ERR, and PIXELDQ arrays are unchanged.

Step Arguments

The `rscd` correction has no step-specific arguments.

Reference Files

The `rscd` correction step uses an RSCD reference file.

RSCD Reference File

REFTYPE
RSCD

Data model
`RSCDModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.RSCDModel.html#jwst.datamodels.RSCDM

The RSCD reference file contains coefficients used to compute the correction.

Reference Selection Keywords for RSCD

CRDS selects appropriate RSCD references based on the following keywords. RSCD is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for RSCD

In addition to the standard reference file keywords listed above, the following keywords are *required* in RSCD reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for RSCD*):

Keyword	Data Model Name
DETECTOR	model.meta.instrument.detector

Reference File Format

RSCD reference files are FITS format, with 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The BINTABLE extension uses the identifier EXTNAME = "RSCD" and the characteristics of the table columns are as follows:

Column name	Data type	Notes
subarray	char*13	FULL or subarray name
readpatt	char*4	SLOW or FAST
rows	char*4	EVEN or ODD
tau	float32	e-folding time scale, in units of frames
ascale	float32	$b1$ in equation 2
pow	float32	$b2$ in equation 2
illum_zp	float32	$illum_{zpt}$ in equation 2.1
illum_slope	float32	$illum_{slope}$ in equation 2.1
illum2	float32	$illum2$ in equation 2.1
param3	float32	$b3$ in equation 2
crossopt	float32	$CrossoverPoint$ in equation 2.2
sat_zp	float32	sat_{zp} in equation 3.1
sat_slope	float32	sat_{slope} in equation 3.1
sat_2	float32	sat_2 in equation 3.1
sat_mzp	float32	sat_{mzp} in equation 3
sat_rowterm	float32	$evenrow_{corrections}$ in equation 3.1
sat_scale	float32	$scale_{\text{sat}}$ in equation 3

The entries in the first 3 columns of the table are used as row selection criteria, matching the exposure properties and row type of the data. The remaining 14 columns contain the parameter values for the double-exponential correction function.

The general form of the correction to be added to the input data is:

```
corrected data = input data + dn_accumulated * scale * exp(-T / tau)  (Equation 1)
```

where:

- T is the time since the last group in the previous integration
- tau is the exponential time constant found in the RSCD table
- dn_accumulated is the DN level that was accumulated for the pixel from the previous integration.

In cases where the last integration does not saturate, the $scale$ term in equation 1 is determined according to:

$$scale = b1 * [Counts2^{b2} * [1/exp(Counts2/b3) - 1] \quad (Equation\ 2)$$

The following two additional equations are used in Equation 2:

$$b1 = ascale * (illum_{zpt} + illum_{slope} * N + illum2 * N^2) \quad (Equation\ 2.1)$$

$$Counts2 = Final\ DN\ in\ the\ last\ group\ in\ the\ previous\ integration - Crossover\ Point \quad (Equation\ 2.2)$$

where:

- N in equation 2.1 is the number of groups per integration
- Crossover Point in equation 2.2 is column CROSSOPT in the RSCD table.

If the previous integration saturates, $scale$ is no longer calculated using equations 2 - 2.2. Instead it is calculated using equations 3 and 3.1:

$$scale_{\text{sat}} = slope * Counts3 + sat_{\text{mzp}} \quad (Equation\ 3)$$

$$slope = sat_{zp} + sat_{slope} * N + sat_2 * N^2 + evenrow_{corrections} \quad (Equation\ 3.1)$$

where:

- $Counts3$ is an estimate of what the last group in the previous integration would have been if saturation did not exist
- $scale_{\text{sat}}$ is sat_scale in the RSCD table
- sat_{mzp} is sat_mzp in the RSCD table
- sat_{zp} is sat_zp in the RSCD table
- sat_{slope} is sat_slope in the RSCD table
- sat_2 is sat2 in the RSCD table
- $evenrow_{corrections}$ is sat_rowterm in the RSCD table
- N is the number of groups per integration

jwst.rscd Package

Classes

RscdStep([name, parent, config_file, ...])	RscdStep: Performs an RSCD correction to MIRI data.

RscdStep

class jwst.rscd.**RscdStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

RscdStep: Performs an RSCD correction to MIRI data. Baseline version flags the first N groups as 'DO_NOT_USE' in the 2nd and later integrations in a copy of the input science data model. Enhanced version is not ready nor enabled.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.

- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'rscd'

reference_file_types = ['rscd']

spec

```
type = option('baseline','enhanced',default = 'baseline') # Type of correction
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.53 Saturation Detection

Description

Class
jwst.saturation.SaturationStep

Alias
saturation

The core algorithm for this step is called from the external package `stcal`, an STScI effort to unify common calibration processing algorithms for use by multiple observatories.

Saturation Checking

The `saturation` step flags pixels at or below the A/D floor or above the saturation threshold. Pixels values are flagged as saturated if the pixel value is larger than the defined saturation threshold. Pixel values are flagged as below the A/D floor if they have a value of zero DN.

This step loops over all integrations within an exposure, examining each one group-by-group, comparing the pixel values in the SCI array with defined saturation thresholds for each pixel. When it finds a pixel value in a given group that is above the saturation threshold (high saturation), it sets the "SATURATED" flag in the corresponding location of the "GROUPDQ" array in the science exposure. When it finds a pixel in a given group that has a zero or negative value (below the A/D floor), it sets the "AD_FLOOR" and "DO_NOT_USE" flags in the corresponding location of the "GROUPDQ" array in the science exposure. For the saturation case, it also flags all subsequent groups for that pixel as saturated. For example, if there are 10 groups in an integration and group 7 is the first one to cross the saturation threshold for a given pixel, then groups 7 through 10 will all be flagged for that pixel.

Pixels with thresholds set to NaN or flagged as "NO_SAT_CHECK" in the saturation reference file have their thresholds set above the 16-bit A-to-D converter limit of 65535 and hence will never be flagged as saturated. The "NO_SAT_CHECK" flag is propagated to the PIXELDQ array in the output science data to indicate which pixels fall into this category.

Charge Migration

There is an effect in IR detectors that results in charge migrating (spilling) from a pixel that has "hard" saturation (i.e. where the pixel no longer accumulates charge) into neighboring pixels. This results in non-linearities in the accumulating signal ramp in the neighboring pixels and hence the ramp data following the onset of saturation is not usable.

The `saturation` step accounts for charge migration by flagging - as saturated - all pixels neighboring a pixel that goes above the saturation threshold. This is accomplished by first flagging all pixels that cross their saturation thresholds and then making a second pass through the data to flag neighbors within a specified region. The region of neighboring pixels is specified as a 2N+1 pixel wide box that is centered on the saturating pixel and N is set by the step parameter `n_pix_grow_sat`. The default value is 1, resulting in a 3x3 box of neighboring pixels that will be flagged.

NIRSpec IRS2 Readouts

NIRSpec data acquired using the "IRS2" readout pattern require special handling in this step, due to the extra reference pixel values that are interleaved within the science data. The saturation reference file data does not contain extra entries for these pixels. The step-by-step process is as follows:

1. Retrieve and load data from the appropriate "SATURATION" reference file from CRDS
2. If the input science exposure used the NIRSpec IRS2 readout pattern:
    - Create a temporary saturation array that is the same size as the IRS2 readout
    - Copy the saturation threshold values from the original reference data into the larger saturation array, skipping over the interleaved reference pixel locations within the array
3. If the input science exposure used a subarray readout, extract the matching subarray from the full-frame saturation reference file data
4. For pixels that contain NaN in the reference file saturation threshold array or are flagged in the reference file with "NO_SAT_CHECK" (no saturation check available), propagate the "NO_SAT_CHECK" flag to the science data PIXELDQ array
5. For each group in the input science data, set the "SATURATION" flag in the "GROUPDQ" array if the pixel value is greater than or equal to the saturation threshold from the reference file

NIRCam Frame 0

If the input contains a frame zero data cube, the frame zero image for each integration is checked for saturation in the same way as the regular science data. This means doing the same comparison of pixel values in the frame zero image to the saturation thresholds defined in the saturation reference file. Because the frame zero does not carry its own Data Quality (DQ) information, pixels found to be above the saturation threshold are simply reset to a value of zero in the frame zero image itself. Subsequent calibration steps are setup to recognize these zero values as indicating that the data were saturated.

Subarrays

The `saturation` step will accept either full-frame or subarray saturation reference files. If only a full-frame reference file is available, the step will extract a subarray to match that of the science exposure. Otherwise, subarray-specific saturation reference files will be used if they are available.

Step Arguments

The `saturation` step has one optional argument:

`--n_pix_grow_sat` (integer, default=1)
: The distance to use when growing saturation flag values to neighboring pixels, in order to account for charge migration (spilling). The total region size is 2*n_pix_grow_sat+1 pixels, centered on the primary pixel.

Reference Files

The `saturation` step uses a SATURATION reference file.

SATURATION Reference File

REFTYPE
SATURATION

Data model
`jwst.datamodels.SaturationModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.SaturationModel

The SATURATION reference file contains pixel-by-pixel saturation threshold values.

Reference Selection Keywords for SATURATION

CRDS selects appropriate SATURATION references based on the following keywords. SATURATION is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
MIRI	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, SUBARRAY, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for SATURATION

In addition to the standard reference file keywords listed above, the following keywords are *required* in SATURATION reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for SATURATION*):

Keyword	Data Model Name	Instrument
DETECTOR	model.meta.instrument.detector	All
SUBARRAY	model.meta.subarray.name	NIRSpec

Reference File Format

SATURATION reference files are FITS format, with 2 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	integer
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The values in the `SCI` array give the saturation threshold in units of DN for each pixel.

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

jwst.saturation Package

Classes

SaturationStep([name, parent, config_file, ...])	This Step sets saturation flags.

SaturationStep

class jwst.saturation.**SaturationStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

This Step sets saturation flags.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'saturation'

reference_file_types = ['saturation']

spec

```
n_pix_grow_sat = integer(default=1) # number of layers adjacent pixels to flag
```

Methods Documentation

`process`(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.54 SkyMatch

Description

Class
`jwst.skymatch.SkymatchStep`

Alias
skymatch

Overview

The `skymatch` step can be used to compute sky values in a collection of input images that contain both sky and source signal. The sky values can be computed for each image separately or in a way that matches the sky levels amongst the collection of images so as to minimize their differences. This operation is typically applied before doing cosmic-ray rejection and combining multiple images into a mosaic. When running the `skymatch` step in a matching mode, it compares *total* signal levels in *the overlap regions* of a set of input images and computes the signal offsets for each image that will minimize – in a least squares sense – the residuals across the entire set. This comparison is performed directly on the input images without resampling them onto a common grid. The overlap regions are computed directly on the sky (celestial sphere) for each pair of input images. Matching based on total signal level is especially useful for images that are dominated by large, diffuse sources, where it is difficult – if not impossible – to find and measure true sky.

Note that the meaning of "sky background" depends on the chosen sky computation method. When the matching method is used, for example, the reported "sky" value is only the offset in levels between images and does not necessarily include the true total sky level.

Note: Throughout this document the term "sky" is used in a generic sense, referring to any kind of non-source background signal, which may include actual sky, as well as instrumental (e.g. thermal) background, etc.

The step records information in three keywords that are included in the output files:

BKGMETH
records the sky method that was used to compute sky levels

BKGLEVEL
the sky level computed for each image

BKGSUB
a boolean indicating whether or not the sky was subtracted from the output images. Note that by default the step argument "subtract" is set to `False`, which means that the sky will *NOT* be subtracted (see the *skymatch step arguments* for more details).

Both the "BKGSUB" and "BKGLEVEL" keyword values are important information for downstream tasks, such as *outlier detection* and *resampling*. Outlier detection will use the BKGLEVEL values to internally equalize the images, which is necessary to prevent false detections due to overall differences in signal levels between images, and the resample step will subtract the BKGLEVEL values from each input image when combining them into a mosaic.

Assumptions

When matching sky background, the code needs to compute bounding polygon intersections in world coordinates. The input images, therefore, need to have a valid WCS, generated by the *assign_wcs* step.

Algorithms

The `skymatch` step provides several methods for constant sky background value computations.

The first method, called "local", essentially is an enhanced version of the original sky subtraction method used in older versions of astrodrizzle (https://drizzlepac.readthedocs.io/en/latest/astrodrizzle.html). This method simply computes the mean/median/mode/etc. value of the sky separately in each input image. This method was upgraded to be able to use DQ flags to remove bad pixels from being used in the computations of sky statistics.

In addition to the classic "local" method, two other methods have been introduced: "global" and "match", as well as a combination of the two – "global+match".

1. The "global" method essentially uses the "local" method to first compute a sky value for each image separately, and then assigns the minimum of those results to all images in the collection. Hence after subtraction of the sky values only one image will have a net sky of zero, while the remaining images will have some small positive residual.

2. The "match" algorithm computes only a correction value for each image, such that, when applied to each image, the mismatch between *all* pairs of images is minimized, in the least-squares sense. For each pair of images, the sky mismatch is computed *only* in the regions in which the two images overlap on the sky.

   This makes the "match" algorithm particularly useful for equalizing sky values in large mosaics in which one may have only pair-wise intersection of adjacent images without having a common intersection region (on the sky) in all images.

   Note that if the argument "match_down=True", matching will be done to the image with the lowest sky value, and if "match_down=False" it will be done to the image with the highest value (see *skymatch step arguments* for full details).

3. The "global+match" algorithm combines the "global" and "match" methods. It uses the "global" algorithm to find a baseline sky value common to all input images and the "match" algorithm to equalize sky values among images. The direction of matching (to the lowest or highest) is again controlled by the "match_down" argument.

In the "local" and "global" methods, which find sky levels in each image, the calculation of the image statistics takes advantage of sigma clipping to remove contributions from isolated sources. This can work well for accurately determining the true sky level in images that contain semi-large regions of empty sky. The "match" algorithm, on the other

hand, compares the *total* signal levels integrated over regions of overlap in each image pair. This method can produce better results when there are no large empty regions of sky in the images. This method cannot measure the true sky level, but instead provides additive corrections that can be used to equalize the signal between overlapping images.

Examples

To get a better idea of the behavior of these different methods, the tables below show the results for two hypothetical sets of images. The first example is for a set of 6 images that form a 2x3 mosaic, with every image having overlap with its immediate neighbors. The first column of the table gives the actual (fake) sky signal that was imposed in each image, and the subsequent columns show the results computed by each method (i.e. the values of the resulting BKGLEVEL keywords). All results are for the case where the step argument `match_down = True`, which means matching is done to the image with the lowest sky value. Note that these examples are for the highly simplistic case where each example image contains nothing but the constant sky value. Hence the sky computations are not affected at all by any source content and are therefore able to determine the sky values exactly in each image. Results for real images will of course not be so exact.

Sky	Local	Global	Match	Global+Match
100	100	100	0	100
120	120	100	20	120
105	105	100	5	105
110	110	100	10	110
105	105	100	5	105
115	115	100	15	115

local
: finds the sky level of each image independently of the rest.

global
: uses the minimum sky level found by "local" and applies it to all images.

match
: with "match_down=True" finds the offset needed to match all images to the level of the image with the lowest sky level.

global+match
: with "match_down=True" finds the offsets and global value needed to set all images to a sky level of zero. In this trivial example, the results are identical to the "local" method.

The second example is for a set of 7 images, where the first 4 form a 2x2 mosaic, with overlaps, and the second set of 3 images forms another mosaic, with internal overlap, but the 2 mosaics do *NOT* overlap one another.

Sky	Local	Global	Match	Global+Match
100	100	90	0	86.25
120	120	90	20	106.25
105	105	90	5	91.25
110	110	90	10	96.25
95	95	90	8.75	95
90	90	90	3.75	90
100	100	90	13.75	100

In this case, the "local" method again computes the sky in each image independently of the rest, and the "global" method sets the result for each image to the minimum value returned by "local". The matching results, however, require some explanation. With "match" only, all of the results give the proper offsets required to equalize the images contained within

each mosaic, but the algorithm does not have the information needed to match the two (non-overlapping) mosaics to one another. Similarly, the "global+match" results again provide proper matching within each mosaic, but will leave an overall residual in one of the mosaics.

Limitations and Discussions

As aluded to above, the best sky computation method depends on the nature of the data in the input images. If the input images contain mostly compact, isolated sources, the "local" and "global" algorithms can do a good job at finding the true sky level in each image. If the images contain large, diffuse sources, the "match" algorithm is more appropriate, assuming of course there is sufficient overlap between images from which to compute the matching values. In the event there is not overlap between all of the images, as illustrated in the second example above, the "match" method can still provide useful results for matching the levels within each non-contigous region covered by the images, but will not provide a good overall sky level across all of the images. In these situations it is more appropriate to either process the non-contiguous groups independently of one another or use the "local" or "global" methods to compute the sky separately in each image. The latter option will of course only work well if the images are not domimated by extended, diffuse sources.

The primary reason for introducing the `skymatch` algorithm was to try to equalize the sky in large mosaics in which computation of the absolute sky is difficult, due to the presence of large diffuse sources in the image. As discussed above, the `skymatch` step accomplishes this by comparing the sky values in the overlap regions of each image pair. The quality of sky matching will obviously depend on how well these sky values can be estimated. True background may not be present at all in some images, in which case the computed "sky" may be the surface brightness of a large galaxy, nebula, etc.

Here is a brief list of possible limitations and factors that can affect the outcome of the matching (sky subtraction in general) algorithm:

1. Because sky computation is performed on *flat-fielded* but *not distortion corrected* images, it is important to keep in mind that flat-fielding is performed to obtain correct surface brightnesses. Because the surface brightness of a pixel containing a point-like source will change inversely with a change to the pixel area, it is advisable to mask point-like sources through user-supplied mask files. Values different from zero in user-supplied masks indicate good data pixels. Alternatively, one can use the `upper` parameter to exclude the use of pixels containing bright objects when performing the sky computations.

2. The input images may contain cosmic rays. This algorithm does not perform CR cleaning. A possible way of minimizing the effect of the cosmic rays on sky computations is to use clipping (`nclip` > 0) and/or set the `upper` parameter to a value larger than most of the sky background (or extended sources) but lower than the values of most CR-affected pixels.

3. In general, clipping is a good way of eliminating bad pixels: pixels affected by CR, hot/dead pixels, etc. However, for images with complicated backgrounds (extended galaxies, nebulae, etc.), affected by CR and noise, the clipping process may mask different pixels in different images. If variations in the background are too strong, clipping may converge to different sky values in different images even when factoring in the true difference in the sky background between the two images.

4. In general images can have different true background values (we could measure it if images were not affected by large diffuse sources). However, arguments such as `lower` and `upper` will apply to all images regardless of the intrinsic differences in sky levels (see *skymatch step arguments*).

Step Arguments

The `skymatch` step uses the following optional arguments:

General sky matching parameters:

`skymethod` (str, default='match')
The sky computation algorithm to be used. Allowed values: `local`, `global`, `match`, `global+match`

`match_down` (boolean, default=True)
Specifies whether the sky *differences* should be subtracted from images with higher sky values (`match_down` = `True` (https://docs.python.org/3/library/constants.html#True)) in order to match the image with the lowest sky, or sky differences should be added to the images with lower sky values to match the sky of the image with the highest sky value (`match_down` = `False` (https://docs.python.org/3/library/constants.html#False)). **NOTE**: this argument only applies when `skymethod` is either `match` or `global+match`.

`subtract` (boolean, default=False)
Specifies whether the computed sky background values are to be subtracted from the images. The BKGSUB keyword (boolean) will be set in each output image to record whether or not the background was subtracted.

Image bounding polygon parameters:

`stepsize` (int, default=None)
Spacing between vertices of the images bounding polygon. The default value of `None` (https://docs.python.org/3/library/constants.html#None) creates bounding polygons with four vertices corresponding to the corners of the image.

Sky statistics parameters:

`skystat` (str, default='mode')
Statistic to be used for sky background computations. Supported values are: `mean`, `mode`, `midpt`, and `median`.

`dqbits` (str, default='~DO_NOT_USE+NON_SCIENCE')
The DQ bit values from the input images' DQ arrays that should be considered "good" when building masks for sky computations. See DQ flag *Parameter Specification* for details. The default value rejects pixels flagged as either 'DO_NOT_USE' or 'NON_SCIENCE' and considers all others to be good.

`lower` (float, default=None)
An optional value indicating the lower limit of usable pixel values for computing the sky. This value should be specified in the units of the input images.

`upper` (float, default=None)
An optional value indicating the upper limit of usable pixel values for computing the sky. This value should be specified in the units of the input images.

`nclip` (int, default=5)
The number of clipping iterations to use when computing sky values.

`lsig` (float, default=4.0)
Lower clipping limit, in sigma, used when computing the sky value.

`usig` (float, default=4.0)
Upper clipping limit, in sigma, used when computing the sky value.

`binwidth` (float, default=0.1)
Bin width, in sigma, used to sample the distribution of pixel values in order to compute the sky background using statistics that require binning, such as `mode` and `midpt`.

Reference File

The skymatch step does not use any reference files.

Also See:

skymatch_step

The skymatch_step function (class name SkyMatchStep) is the top-level function used to call the skymatch operation from the JWST calibration pipeline.

jwst.skymatch.skymatch_step Module

JWST pipeline step for sky matching.

Authors
Mihai Cara

Classes

SkyMatchStep(*args, **kwargs)	SkyMatchStep: Subtraction or equalization of sky background in science images.

SkyMatchStep

class jwst.skymatch.skymatch_step.**SkyMatchStep**(**args*, ***kwargs*)

Bases: JwstStep

SkyMatchStep: Subtraction or equalization of sky background in science images.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'skymatch'

reference_file_types = []

spec

```
# General sky matching parameters:
skymethod = option('local', 'global', 'match', 'global+match', default='match')
↪# sky computation method
match_down = boolean(default=True) # adjust sky to lowest measured value?
subtract = boolean(default=False) # subtract computed sky from image data?

# Image's bounding polygon parameters:
stepsize = integer(default=None) # Max vertex separation

# Sky statistics parameters:
skystat = option('median', 'midpt', 'mean', 'mode', default='mode') # sky␣
↪statistics
dqbits = string(default='~DO_NOT_USE+NON_SCIENCE') # "good" DQ bits
lower = float(default=None) # Lower limit of "good" pixel values
upper = float(default=None) # Upper limit of "good" pixel values
nclip = integer(min=0, default=5) # number of sky clipping iterations
lsigma = float(min=0.0, default=4.0) # Lower clipping limit, in sigma
usigma = float(min=0.0, default=4.0) # Upper clipping limit, in sigma
binwidth = float(min=0.0, default=0.1) # Bin width for 'mode' and 'midpt'␣
↪`skystat`, in sigma
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

skymatch

The `skymatch` function performs the actual sky matching operations on the input image data models.

jwst.skymatch.skymatch Module

A module that provides functions for matching sky in overlapping images.

Authors

Mihai Cara

Functions

match(images[, skymethod, match_down, subtract])	A function to compute and/or "equalize" sky background in input images.

match

jwst.skymatch.skymatch.**match**(*images*, *skymethod='global+match'*, *match_down=True*, *subtract=False*)

A function to compute and/or "equalize" sky background in input images.

Note: Sky matching ("equalization") is possible only for **overlapping** images.

Parameters

- **images** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of SkyImage or SkyGroup*) – A list of of `SkyImage` or `SkyGroup` objects.
- **skymethod** (*{'local', 'global+match', 'global', 'match'}, optional*) – Select the algorithm for sky computation:

– **'local'** : compute sky background values of each input image or group of images (members of the same "exposure"). A single sky value is computed for each group of images.

Note: This setting is recommended when regions of overlap between images are dominated by "pure" sky (as opposed to extended, diffuse sources).

– **'global'** : compute a common sky value for all input images and groups of images. With this setting `local` will compute sky values for each input image/group, find the minimum sky value, and then it will set (and/or subtract) the sky value of each input image to this minimum value. This method *may* be useful when the input images have been already matched.

– **'match'** : compute differences in sky values between images and/or groups in (pair-wise) common sky regions. In this case the computed sky values will be relative (delta) to the sky computed in one of the input images whose sky value will be set to (reported to be) 0. This setting will "equalize" sky values between the images in large mosaics. However, this method is not recommended when used in conjunction with astrodrizzle (http://stsdas.stsci.edu/stsci_python_sphinxdocs_2.13/drizzlepac/astrodrizzle.html) because it computes relative sky values while `astrodrizzle` needs "absolute" sky values for median image generation and CR rejection.

– **'global+match'** : first use the **'match'** method to equalize sky values between images and then find a minimum "global" sky value amongst all input images.

Note: This is the *recommended* setting for images containing diffuse sources (e.g., galaxies, nebulae) covering significant parts of the image.

- **match_down** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Specifies whether the sky *differences* should be subtracted from images with higher sky values (`match_down` = `True` (https://docs.python.org/3/library/constants.html#True)) to match the image with the lowest sky or sky differences should be added to the images with lower sky values to match the sky of the image with the highest sky value (`match_down` = `False` (https://docs.python.org/3/library/constants.html#False)).

Note: This setting applies *only* when the `skymethod` parameter is either `'match'` or `'global+match'`.

- **subtract** (*bool* (https://docs.python.org/3/library/functions.html#bool) *(Default = False)*) – Subtract computed sky value from image data.

Raises

`TypeError` (https://docs.python.org/3/library/exceptions.html#TypeError) – The `images` argument must be a Python list of `SkyImage` and/or `SkyGroup` objects.

Notes

match() provides new algorithms for sky value computations and enhances previously available algorithms used by, e.g., astrodrizzle (http://stsdas.stsci.edu/stsci_python_sphinxdocs_2.13/drizzlepac/astrodrizzle.html).

Two new methods of sky subtraction have been introduced (compared to the standard `'local'`): `'global'` and `'match'`, as well as a combination of the two – `'global+match'`.

- The `'global'` method computes the minimum sky value across *all* input images and/or groups. That sky value is then considered to be the background in all input images.
- The `'match'` algorithm is somewhat similar to the traditional sky subtraction method (`skymethod = 'local'`) in the sense that it measures the sky independently in input images (or groups). The major differences are that, unlike the traditional method,

  1. `'match'` algorithm computes *relative* (delta) sky values with regard to the sky in a reference image chosen from the input list of images; *and*
  2. Sky statistics are computed only in the part of the image that intersects other images.

  This makes the `'match'` sky computation algorithm particularly useful for "equalizing" sky values in large mosaics in which one may have only (at least) pair-wise intersection of images without having a common intersection region (on the sky) in all images.

  The `'match'` method works in the following way: for each pair of intersecting images, an equation is written that requires that average surface brightness in the overlapping part of the sky be equal in both images. The final system of equations is then solved for unknown background levels.

  > **Warning:** The current algorithm is not capable of detecting cases where some subsets of intersecting images (from the input list of images) do not intersect at all with other subsets of intersecting images (except for the simple case when *single* images do not intersect any other images). In these cases the algorithm will find equalizing sky values for each intersecting subset of images and/or groups of images. However since these subsets of images do not intersect each other, sky will be matched only within each subset and the "inter-subset" sky mismatch could be significant.
  >
  > Users are responsible for detecting such cases and adjusting processing accordingly.

- The `'global+match'` algorithm combines the `'match'` and `'global'` methods in order to overcome the limitation of the `'match'` method described in the note above: it uses the `'global'` algorithm to find a baseline sky value common to all input images and the `'match'` algorithm to "equalize" sky values in the mosaic. Thus, the sky value of the "reference" image will be equal to the baseline sky value (instead of 0 in `'match'` algorithm alone).

Remarks:

- *match()* works directly on *geometrically distorted* flat-fielded images thus avoiding the need to perform distortion correction on the input images.

  Initially, the footprint of a chip in an image is approximated by a 2D planar rectangle representing the borders of chip's distorted image. After applying distortion model to this rectangle and projecting it onto the celestial sphere, it is approximated by spherical polygons. Footprints of exposures and mosaics are computed as unions of such spherical polygons while overlaps of image pairs are found by intersecting these spherical polygons.

Limitations and Discussions:

Primary reason for introducing "sky match" algorithm was to try to equalize the sky in large mosaics in which computation of the "absolute" sky is difficult due to the presence of large diffuse sources in the image. As discussed above, *match()* accomplishes this by comparing "sky values" in a pair of images

in the overlap region (that is common to both images). Quite obviously the quality of sky "matching" will depend on how well these "sky values" can be estimated. We use quotation marks around *sky values* because for some image "true" background may not be present at all and the measured sky may be the surface brightness of large galaxy, nebula, etc.

In the discussion below we will refer to parameter names in `SkyStats` and these parameter names may differ from the parameters of the actual `skystat` object passed to initializer of the `SkyImage`.

Here is a brief list of possible limitations/factors that can affect the outcome of the matching (sky subtraction in general) algorithm:

- Since sky subtraction is performed on *flat-fielded* but *not distortion corrected* images, it is important to keep in mind that flat-fielding is performed to obtain uniform surface brightness and not flux. This distinction is important for images that have not been distortion corrected. As a consequence, it is advisable that point-like sources be masked through the user-supplied mask files. Values different from zero in user-supplied masks indicate "good" data pixels. Alternatively, one can use `upper` parameter to limit the use of bright objects in sky computations.
- Normally, distorted flat-fielded images contain cosmic rays. This algorithm does not perform CR cleaning. A possible way of minimizing the effect of the cosmic rays on sky computations is to use clipping (`nclip` > 0) and/or set `upper` parameter to a value larger than most of the sky background (or extended source) but lower than the values of most CR pixels.
- In general, clipping is a good way of eliminating "bad" pixels: pixels affected by CR, hot/dead pixels, etc. However, for images with complicated backgrounds (extended galaxies, nebulae, etc.), affected by CR and noise, clipping process may mask different pixels in different images. If variations in the background are too strong, clipping may converge to different sky values in different images even when factoring in the "true" difference in the sky background between the two images.
- In general images can have different "true" background values (we could measure it if images were not affected by large diffuse sources). However, arguments such as `lower` and `upper` will apply to all images regardless of the intrinsic differences in sky levels.

skyimage

jwst.skymatch.skyimage Module

The `skyimage` module contains algorithms that are used by `skymatch` to manage all of the information for footprints (image outlines) on the sky as well as perform useful operations on these outlines such as computing intersections and statistics in the overlap regions.

Authors

Mihai Cara (contact: help@stsci.edu)

Classes

SkyImage(image, wcs_fwd, wcs_inv[, ...])	Container that holds information about properties of a *single* image such as:
SkyGroup(images[, id, sky])	Holds multiple SkyImage objects whose sky background values must be adjusted together.
DataAccessor()	Base class for all data accessors.
NDArrayInMemoryAccessor(data)	Acessor for in-memory numpy.ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy data.
NDArrayMappedAccessor(data[, tmpfile, ...])	Data accessor for arrays stored in temporary files.

SkyImage

class jwst.skymatch.skyimage.**SkyImage**(*image*, *wcs_fwd*, *wcs_inv*, *pix_area=1.0*, *convf=1.0*, *mask=None*, *id=None*, *skystat=None*, *stepsize=None*, *meta=None*, *reduce_memory_usage=True*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Container that holds information about properties of a *single* image such as:

- image data;
- WCS of the chip image;
- bounding spherical polygon;
- id;
- pixel area;
- sky background value;
- sky statistics parameters;
- mask associated image data indicating "good" (1) data.

Initializes the SkyImage object.

Parameters

- **image** (*numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray), *NDArrayDataAccessor*) – A 2D array of image data or a NDArrayDataAccessor.
- **wcs_fwd** (*function*) – "forward" pixel-to-world transformation function.
- **wcs_inv** (*function*) – "inverse" world-to-pixel transformation function.
- **pix_area** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Average pixel's sky area.
- **convf** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Conversion factor that when multiplied to *image* data converts the data to "uniform" (across multiple images) surface brightness units.

  Note: The functionality to support this conversion is not yet implemented and at this moment convf is ignored.

- **mask** (*numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray), *NDArrayDataAccessor*) – A 2D array or `NDArrayDataAccessor` of a 2D array that indicates which pixels in the input *image* should be used for sky computations (`1`) and which pixels should **not** be used for sky computations (`0`).
- **id** (*anything*) – The value of this parameter is simple stored within the `SkyImage` object. While it can be of any type, it is preferable that *id* be of a type with nice string representation.
- **skystat** (*callable, None, optional*) – A callable object that takes a either a 2D image (2D `numpy.ndarray` (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray)) or a list of pixel values (a Nx1 array) and returns a tuple of two values: some statistics (e.g., mean, median, etc.) and number of pixels/values from the input image used in computing that statistics.

  When *skystat* is not set, `SkyImage` will use `SkyStats` object to perform sky statistics on image data.
- **stepsize** (*int* (https://docs.python.org/3/library/functions.html#int), *None, optional*) – Spacing between vertices of the image's bounding polygon. Default value of `None` (https://docs.python.org/3/library/constants.html#None) creates bounding polygons with four vertices corresponding to the corners of the image.
- **meta** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict), *None, optional*) – A dictionary of various items to be stored within the `SkyImage` object.
- **reduce_memory_usage** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Indicates whether to attempt to minimize memory usage by attaching input `image` and/or `mask` `numpy.ndarray` (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) arrays to file-mapped accessor. This has no effect when input parameters `image` and/or `mask` are already of `NDArrayDataAccessor` objects.

Attributes Summary

id	Set or get SkyImage's *id*.
image	Set or get SkyImage's image data array.
image_shape	Get SkyImage's image data shape.
is_sky_valid	Indicates whether sky value was successfully computed.
mask	Set or get SkyImage's mask data array or None (https://docs.python.org/3/library/constants.html#None).
pix_area	Set or get mean pixel area.
poly_area	Get bounding polygon area in srad units.
polygon	Get image's bounding polygon.
radec	Get RA and DEC of the vertices of the bounding polygon as a ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#nu of shape (N, 2) where N is the number of vertices + 1.
sky	Sky background value.
skystat	Stores/retrieves a callable object that takes a either a 2D image (2D numpy.ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#nu or a list of pixel values (a Nx1 array) and returns a tuple of two values: some statistics (e.g., mean, median, etc.) and number of pixels/values from the input image used in computing that statistics.

Methods Summary

calc_bounding_polygon([stepsize])	Compute image's bounding polygon.
calc_sky([overlap, delta])	Compute sky background value.
copy()	Return a shallow copy of the SkyImage object.
intersection(skyimage)	Compute intersection of this SkyImage object and another SkyImage, SkyGroup, or SphericalPolygon object.
set_builtin_skystat([skystat, lower, upper, ...])	Replace already set *skystat* with a "built-in" version of a statistics callable object used to measure sky background.

Attributes Documentation

id

Set or get SkyImage's *id*.

While *id* can be of any type, it is preferable that *id* be of a type with nice string representation.

image

Set or get SkyImage's image data array.

image_shape

Get SkyImage's image data shape.

is_sky_valid

Indicates whether sky value was successfully computed. Must be set externally.

mask

Set or get SkyImage's mask data array or None (https://docs.python.org/3/library/constants.html#None).

pix_area

Set or get mean pixel area.

poly_area

Get bounding polygon area in srad units.

polygon

Get image's bounding polygon.

radec

Get RA and DEC of the vertices of the bounding polygon as a ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) of shape (N, 2) where N is the number of vertices + 1.

sky

Sky background value. See *calc_sky* for more details.

skystat

Stores/retrieves a callable object that takes a either a 2D image (2D numpy.ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray)) or a list of pixel values (a Nx1 array) and returns a tuple of two values: some statistics (e.g., mean, median, etc.) and number of pixels/values from the input image used in computing that statistics.

When *skystat* is not set, SkyImage will use SkyStats object to perform sky statistics on image data.

Methods Documentation

calc_bounding_polygon(*stepsize=None*)

Compute image's bounding polygon.

Parameters

stepsize (*int* (https://docs.python.org/3/library/functions.html#int), *None*, *optional*) – Indicates the maximum separation between two adjacent vertices of the bounding polygon along each side of the image. Corners of the image are included automatically. If stepsize is None (https://docs.python.org/3/library/constants.html#None), bounding polygon will contain only vertices of the image.

calc_sky(*overlap=None*, *delta=True*)

Compute sky background value.

Parameters

- **overlap** (*SkyImage*, *SkyGroup*, *SphericalPolygon*, *list* (https://docs.python.org/3/library/stdtypes.html#list) *of tuples*, *None*, *optional*) – Another SkyImage, SkyGroup, spherical_geometry.polygons.SphericalPolygon, or a list of tuples of (RA, DEC) of vertices of a spherical polygon. This parameter is used to indicate that sky statistics should computed only in the region of intersection of *this* image with the polygon indicated by overlap. When overlap is None (https://docs.python.org/3/library/constants.html#None), sky statistics will be computed over the entire image.

- **delta** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Should this function return absolute sky value or the difference between the computed value and the value of the sky stored in the *sky* property.

Returns

- **skyval** (*float, None*) – Computed sky value (absolute or relative to the *sky* attribute). If there are no valid data to perform this computations (e.g., because this image does not overlap with the image indicated by `overlap`), `skyval` will be set to `None` (https://docs.python.org/3/library/constants.html#None).
- **npix** (*int*) – Number of pixels used to compute sky statistics.
- **polyarea** (*float*) – Area (in srad) of the polygon that bounds data used to compute sky statistics.

`copy()`

Return a shallow copy of the `SkyImage` object.

`intersection(skyimage)`

Compute intersection of this `SkyImage` object and another `SkyImage`, `SkyGroup`, or `SphericalPolygon` object.

Parameters

skyimage (*SkyImage, SkyGroup, SphericalPolygon*) – Another object that should be intersected with this `SkyImage`.

Returns

polygon – A `SphericalPolygon` that is the intersection of this `SkyImage` and `skyimage`.

Return type

SphericalPolygon

`set_builtin_skystat(skystat='median', lower=None, upper=None, nclip=5, lsigma=4.0, usigma=4.0, binwidth=0.1)`

Replace already set *skystat* with a "built-in" version of a statistics callable object used to measure sky background.

See `SkyStats` for the parameter description.

SkyGroup

`class jwst.skymatch.skyimage.SkyGroup(images, id=None, sky=0.0)`

Bases: `object` (https://docs.python.org/3/library/functions.html#object)

Holds multiple `SkyImage` objects whose sky background values must be adjusted together.

`SkyGroup` provides methods for obtaining bounding polygon of the group of `SkyImage` objects and to compute sky value of the group.

Attributes Summary

id	Set or get SkyImage's *id*.
polygon	Get image's bounding polygon.
radec	Get RA and DEC of the vertices of the bounding polygon as a ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#nu of shape (N, 2) where N is the number of vertices + 1.
sky	Sky background value.

Methods Summary

append(value)	Appends a SkyImage to the group.
calc_sky([overlap, delta])	Compute sky background value.
insert(idx, value)	Inserts a SkyImage into the group.
intersection(skyimage)	Compute intersection of this SkyImage object and another SkyImage, SkyGroup, or SphericalPolygon object.

Attributes Documentation

id

Set or get SkyImage's *id*.

While *id* can be of any type, it is preferable that *id* be of a type with nice string representation.

polygon

Get image's bounding polygon.

radec

Get RA and DEC of the vertices of the bounding polygon as a ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) of shape (N, 2) where N is the number of vertices + 1.

sky

Sky background value. See *calc_sky* for more details.

Methods Documentation

append(*value*)

Appends a SkyImage to the group.

calc_sky(*overlap=None*, *delta=True*)

Compute sky background value.

Parameters

- **overlap** (*SkyImage*, *SkyGroup*, *SphericalPolygon*, *list* (https://docs.python.org/3/library/stdtypes.html#list) *of tuples*, *None*,

optional) – Another SkyImage, SkyGroup, spherical_geometry.polygons.SphericalPolygon, or a list of tuples of (RA, DEC) of vertices of a spherical polygon. This parameter is used to indicate that sky statistics should computed only in the region of intersection of *this* image with the polygon indicated by overlap. When overlap is None (https://docs.python.org/3/library/constants.html#None), sky statistics will be computed over the entire image.

- **delta** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Should this function return absolute sky value or the difference between the computed value and the value of the sky stored in the *sky* property.

Returns

- **skyval** (*float, None*) – Computed sky value (absolute or relative to the *sky* attribute). If there are no valid data to perform this computations (e.g., because this image does not overlap with the image indicated by overlap), skyval will be set to None (https://docs.python.org/3/library/constants.html#None).
- **npix** (*int*) – Number of pixels used to compute sky statistics.
- **polyarea** (*float*) – Area (in srad) of the polygon that bounds data used to compute sky statistics.

insert(*idx*, *value*)

Inserts a SkyImage into the group.

intersection(*skyimage*)

Compute intersection of this SkyImage object and another SkyImage, SkyGroup, or SphericalPolygon object.

Parameters

skyimage (*SkyImage, SkyGroup, SphericalPolygon*) – Another object that should be intersected with this SkyImage.

Returns

intersect_poly – A SphericalPolygon that is the intersection of this SkyImage and skyimage.

Return type

SphericalPolygon

DataAccessor

class jwst.skymatch.skyimage.**DataAccessor**

Bases: ABC (https://docs.python.org/3/library/abc.html#abc.ABC)

Base class for all data accessors. Provides a common interface to access data.

Methods Summary

get_data()	
get_data_shape()	
set_data(data)	Sets data.

Methods Documentation

abstract get_data()

abstract get_data_shape()

abstract set_data(*data*)

Sets data.

Parameters

data (*numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray)) – Data array to be set.

NDArrayInMemoryAccessor

class jwst.skymatch.skyimage.**NDArrayInMemoryAccessor**(*data*)

Bases: DataAccessor

Acessor for in-memory numpy.ndarray (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray) data.

Methods Summary

get_data()	
get_data_shape()	
set_data(data)	Sets data.

Methods Documentation

get_data()

get_data_shape()

set_data(*data*)

Sets data.

Parameters

data (*numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray)) – Data array to be set.

NDArrayMappedAccessor

class jwst.skymatch.skyimage.**NDArrayMappedAccessor**(*data*, *tmpfile=None*, *prefix='tmp_skymatch_'*, *suffix='.npy'*, *tmpdir=''*)

Bases: DataAccessor

Data accessor for arrays stored in temporary files.

Methods Summary

get_data()	
get_data_shape()	
set_data(data)	Sets data.

Methods Documentation

get_data()

get_data_shape()

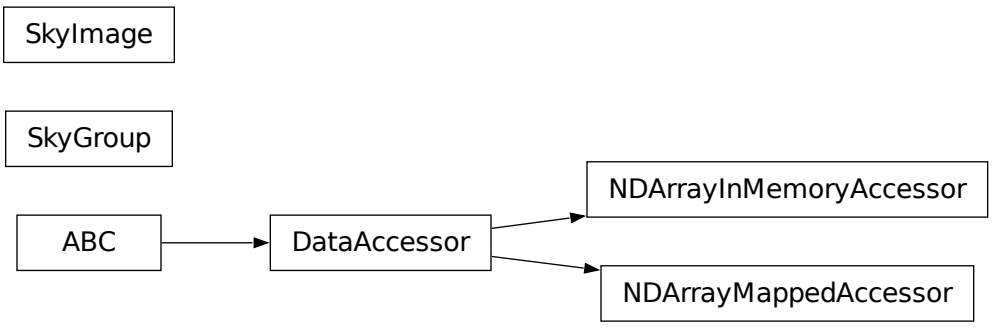
set_data(*data*)

Sets data.

Parameters

data (*numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray)) – Data array to be set.

Class Inheritance Diagram



skystatistics

The `skystatistics` module contains various statistical functions used by `skymatch`.

jwst.skymatch.skystatistics Module

`skystatistics` module provides statistics computation class used by *match()* and `SkyImage`.

Authors

Mihai Cara (contact: help@stsci.edu)

Classes

`SkyStats`([skystat, lower, upper, nclip, ...])	This is a superclass build on top of `stsci.imagestats.ImageStats`.

SkyStats

class jwst.skymatch.skystatistics.**SkyStats**(*skystat='mean'*, *lower=None*, *upper=None*, *nclip=5*, *lsig=4.0*, *usig=4.0*, *binwidth=0.1*, ***kwargs*)

Bases: `object` (https://docs.python.org/3/library/functions.html#object)

This is a superclass build on top of `stsci.imagestats.ImageStats`. Compared to `stsci.imagestats.ImageStats`, `SkyStats` has "persistent settings" in the sense that object's parameters need to be set once and these settings will be applied to all subsequent computations on different data.

Initializes the SkyStats object.

Parameters

- **skystat** (*{'mode', 'median', 'mode', 'midpt'}, optional*) – Sets the statistics that will be returned by *calc_sky*. The following statistics are supported: 'mean', 'mode', 'midpt', and 'median'. First three statistics have the same meaning as in stsdas.toolbox.imgtools.gstatistics (http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?gstatistics) while 'median' will compute the median of the distribution.
- **lower** (*float* (https://docs.python.org/3/library/functions.html#float), *None*, *optional*) – Lower limit of usable pixel values for computing the sky. This value should be specified in the units of the input image(s).
- **upper** (*float* (https://docs.python.org/3/library/functions.html#float), *None*, *optional*) – Upper limit of usable pixel values for computing the sky. This value should be specified in the units of the input image(s).
- **nclip** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – A non-negative number of clipping iterations to use when computing the sky value.
- **lsig** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Lower clipping limit, in sigma, used when computing the sky value.
- **usig** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Upper clipping limit, in sigma, used when computing the sky value.

- **binwidth** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Bin width, in sigma, used to sample the distribution of pixel brightness values in order to compute the sky background statistics.
- **kwargs** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – A dictionary of optional arguments to be passed to `ImageStats`.

Methods Summary

__call__(data)	Call self as a function.
calc_sky(data)	Computes statistics on data.

Methods Documentation

__call__(*data*)

Call self as a function.

calc_sky(*data*)

Computes statistics on data.

Parameters

data (*numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray)) – A numpy array of values for which the statistics needs to be computed.

Returns

statistics – A tuple of two values: (`skyvalue`, `npix`), where `skyvalue` is the statistics specified by the `skystat` parameter during the initialization of the `SkyStats` object and `npix` is the number of pixels used in computing the statistics reported in `skyvalue`.

Return type

tuple (https://docs.python.org/3/library/stdtypes.html#tuple)

Class Inheritance Diagram

region

The `region` module provides a polygon filling algorithm used by `skymatch` to create data masks.

jwst.skymatch.region Module

Polygon filling algorithm.

Classes

`Region`(rid, coordinate_system)	Base class for regions.
`Edge`([name, start, stop, next])	Edge representation
`Polygon`(rid, vertices[, coord_system])	Represents a 2D polygon region with multiple vertices

Region

class jwst.skymatch.region.**Region**(*rid*, *coordinate_system*)

Bases: `object` (https://docs.python.org/3/library/functions.html#object)

Base class for regions.

Parameters

- **rid** (*int* (https://docs.python.org/3/library/functions.html#int) *or string*) – region ID
- **coordinate_system** (*astropy.wcs.CoordinateSystem instance or a string*) – in the context of WCS this would be an instance of wcs.CoordinateSysem

Methods Summary

scan(mask)	Sets mask values to region id for all pixels within the region.

Methods Documentation

scan(*mask*)

Sets mask values to region id for all pixels within the region. Subclasses must define this method.

Parameters
mask (*ndarray*) – a byte array with the shape of the observation to be used as a mask

Returns
mask – pixels which are not included in any region).

Return type
array where the value of the elements is the region ID or 0 (for

Edge

class jwst.skymatch.region.**Edge**(*name=None*, *start=None*, *stop=None*, *next=None*)

Bases: object (https://docs.python.org/3/library/functions.html#object)

Edge representation

An edge has a "start" and "stop" (x,y) vertices and an entry in the GET table of a polygon. The GET entry is a list of these values:

[ymax, x_at_ymin, delta_x/delta_y]

Attributes Summary

next	
start	
stop	
ymax	
ymin	

Methods Summary

compute_AET_entry(edge)	Compute the entry for an edge in the current Active Edge Table
compute_GET_entry()	Compute the entry in the Global Edge Table
intersection(edge)	
is_parallel(edge)	

Attributes Documentation

next

start

stop

ymax

ymin

Methods Documentation

compute_AET_entry(*edge*)

Compute the entry for an edge in the current Active Edge Table

[ymax, x_intersect, 1/m] note: currently 1/m is not used

compute_GET_entry()

Compute the entry in the Global Edge Table

[ymax, x@ymin, 1/m]

intersection(*edge*)

is_parallel(*edge*)

Polygon

class jwst.skymatch.region.**Polygon**(*rid*, *vertices*, *coord_system='Cartesian'*)

Bases: Region

Represents a 2D polygon region with multiple vertices

Parameters

- **rid** (*string*) – polygon id
- **vertices** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *of (x,y) tuples or lists*) – The list is ordered in such a way that when traversed in a counterclockwise direction, the enclosed area is the polygon. The last vertex must coincide with the first vertex, minimum 4 vertices are needed to define a triangle
- **coord_system** (*string*) – coordinate system

Methods Summary

get_edges()	Create a list of Edge objects from vertices
scan(data)	This is the main function which scans the polygon and creates the mask
update_AET(y, AET)	Update the Active Edge Table (AET)

Methods Documentation

get_edges()

Create a list of Edge objects from vertices

scan(*data*)

This is the main function which scans the polygon and creates the mask

Parameters

- **data** (*array*) – the mask array it has all zeros initially, elements within a region are set to the region's ID
- **Algorithm** –

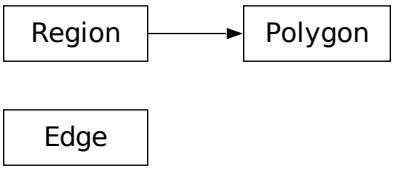
- **(GET)** (*- Set the Global Edge Table*) –
- **GET** (*- Set y to be the smallest y coordinate that has an entry in*) –
- **empty** (*- Initialize the Active Edge Table (AET) to be*) –
- **line** (*- For each scan*) –
  1. Add edges from GET to AET for which ymin==y
  2. Remove edges from AET fro which ymax==y
  3. Compute the intersection of the current scan line with all edges in the AET
  4. Sort on X of intersection point
  5. Set elements between pairs of X in the AET to the Edge's ID

update_AET(*y*, *AET*)

Update the Active Edge Table (AET)

Add edges from GET to AET for which ymin of the edge is equal to the y of the scan line. Remove edges from AET for which ymax of the edge is equal to y of the scan line.

Class Inheritance Diagram



jwst.skymatch Package

This package provides support for sky background subtraction and equalization (matching).

Classes

SkyMatchStep(*args, **kwargs)	SkyMatchStep: Subtraction or equalization of sky background in science images.

SkyMatchStep

class jwst.skymatch.**SkyMatchStep**(**args*, ***kwargs*)

Bases: JwstStep

SkyMatchStep: Subtraction or equalization of sky background in science images.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'skymatch'

reference_file_types = []

spec

```
# General sky matching parameters:
skymethod = option('local', 'global', 'match', 'global+match', default='match')
↪# sky computation method
match_down = boolean(default=True) # adjust sky to lowest measured value?
subtract = boolean(default=False) # subtract computed sky from image data?
```

(continues on next page)

(continued from previous page)

```
# Image's bounding polygon parameters:
stepsize = integer(default=None) # Max vertex separation

# Sky statistics parameters:
skystat = option('median', 'midpt', 'mean', 'mode', default='mode') # sky␣
↪statistics
dqbits = string(default='~DO_NOT_USE+NON_SCIENCE') # "good" DQ bits
lower = float(default=None) # Lower limit of "good" pixel values
upper = float(default=None) # Upper limit of "good" pixel values
nclip = integer(min=0, default=5) # number of sky clipping iterations
lsigma = float(min=0.0, default=4.0) # Lower clipping limit, in sigma
usigma = float(min=0.0, default=4.0) # Upper clipping limit, in sigma
binwidth = float(min=0.0, default=0.1) # Bin width for 'mode' and 'midpt'␣
↪`skystat`, in sigma
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.55 Source Catalog

Description

Class

jwst.source_catalog.SourceCatalogStep

Alias

source_catalog

This step creates a catalog of source photometry and morphologies. Both aperture and isophotal (segment-based) photometry are calculated. Source morphologies are based on 2D image moments within the source segment.

Source Detection

Sources are detected using image segmentation (https://en.wikipedia.org/wiki/Image_segmentation), which is a process of assigning a label to every pixel in an image such that pixels with the same label are part of the same source. The segmentation procedure used is from Photutils source extraction (https://photutils.readthedocs.io/en/latest/segmentation.html). Detected sources must have a minimum number of connected pixels that are each greater than a specified threshold value in an image. The threshold level is usually defined at some multiple of the background standard deviation above the background. The image can also be filtered before thresholding to smooth the noise and maximize the detectability of objects with a shape similar to the filter kernel.

Source Deblending

Overlapping sources are detected as single sources. Separating those sources requires a deblending procedure, such as a multi-thresholding technique used by SExtractor (https://www.astromatic.net/software/sextractor). Here we use the Photutils deblender (https://photutils.readthedocs.io/en/latest/segmentation.html#source-deblending), which is an algorithm that deblends sources using a combination of multi-thresholding and watershed segmentation (https://en.wikipedia.org/wiki/Watershed_(image_processing)). In order to deblend sources, they must be separated enough such that there is a saddle between them.

Source Photometry and Properties

After detecting sources using image segmentation, we can measure their photometry, centroids, and morphological properties. The aperture photometry is measured in three apertures, based on the input encircled energy values. The total aperture-corrected flux and magnitudes are also calculated, based on the largest aperture. Both AB and Vega magnitudes are calculated.

The isophotal photometry is based on photutils segmentation (https://photutils.readthedocs.org/en/latest/segmentation.html). The properties that are currently calculated for each source include source centroids (both in pixel and sky coordinates), isophotal fluxes (and errors), AB and Vega magnitudes (and errors), isophotal area, semimajor and semiminor axis lengths, orientation of the major axis, and sky coordinates at corners of the minimal bounding box enclosing the source.

Photometric errors are calculated from the resampled total-error array contained in the `ERR` (`model.err`) array. Note that this total-error array includes source Poisson noise.

Output Products

Source Catalog Table

The output source catalog table is saved in ECSV format (https://docs.astropy.org/en/stable/io/ascii/ecsv.html).

The table contains a row for each source, with the following default columns (assuming the default encircled energies of 30, 50, and 70):

Column	Description
label	Unique source identification label number
xcentroid	X pixel value of the source centroid (0 indexed)
ycentroid	Y pixel value of the source centroid (0 indexed)
sky_centroid	Sky coordinate of the source centroid

Table 8 – continued from previous page

Column	Description
aper_bkg_flux	The local background value calculated as the sigma-clipped median value in the background annulus apertu
aper_bkg_flux_err	The standard error of the sigma-clipped median background value
aper30_flux	Flux within the 30% encircled energy circular aperture
aper30_flux_err	Flux error within the 30% encircled energy circular aperture
aper50_flux	Flux within the 50% encircled energy circular aperture
aper50_flux_err	Flux error within the 50% encircled energy circular aperture
aper70_flux	Flux within the 70% encircled energy circular aperture
aper70_flux_err	Flux error within the 70% encircled energy circular aperture
aper_total_flux	Total aperture-corrected flux based on the 70% encircled energy circular aperture; should be used only for u
aper_total_flux_err	Total aperture-corrected flux error based on the 70% encircled energy circular aperture; should be used onl
aper30_abmag	AB magnitude within the 30% encircled energy circular aperture
aper30_abmag_err	AB magnitude error within the 30% encircled energy circular aperture
aper50_abmag	AB magnitude within the 50% encircled energy circular aperture
aper50_abmag_err	AB magnitude error within the 50% encircled energy circular aperture
aper70_abmag	AB magnitude within the 70% encircled energy circular aperture
aper70_abmag_err	AB magnitude error within the 70% encircled energy circular aperture
aper_total_abmag	Total aperture-corrected AB magnitude based on the 70% encircled energy circular aperture; should be use
aper_total_abmag_err	Total aperture-corrected AB magnitude error based on the 70% encircled energy circular aperture; should b
aper30_vegamag	Vega magnitude within the 30% encircled energy circular aperture
aper30_vegamag_err	Vega magnitude error within the 30% encircled energy circular aperture
aper50_vegamag	Vega magnitude within the 50% encircled energy circular aperture
aper50_vegamag_err	Vega magnitude error within the 50% encircled energy circular aperture
aper70_vegamag	Vega magnitude within the 70% encircled energy circular aperture
aper70_vegamag_err	Vega magnitude error within the 70% encircled energy circular aperture
aper_total_vegamag	Total aperture-corrected Vega magnitude based on the 70% encircled energy circular aperture; should be us
aper_total_vegamag_err	Total aperture-corrected Vega magnitude error based on the 70% encircled energy circular aperture; should
CI_50_30	Concentration index calculated as (aper50_flux / aper30_flux)
CI_70_50	Concentration index calculated as (aper70_flux / aper50_flux)
CI_70_30	Concentration index calculated as (aper70_flux / aper30_flux)
is_extended	Flag indicating whether the source is extended
sharpness	The DAOFind source sharpness statistic
roundness	The DAOFind source roundness statistic
nn_label	The label number of the nearest neighbor
nn_dist	The distance in pixels to the nearest neighbor
isophotal_flux	Isophotal flux
isophotal_flux_err	Isophotal flux error
isophotal_abmag	Isophotal AB magnitude
isophotal_abmag_err	Isophotal AB magnitude error
isophotal_vegamag	Isophotal Vega magnitude
isophotal_vegamag_err	Isophotal Vega magnitude error
isophotal_area	Isophotal area
semimajor_sigma	1-sigma standard deviation along the semimajor axis of the 2D Gaussian function that has the same second
semiminor_sigma	1-sigma standard deviation along the semiminor axis of the 2D Gaussian function that has the same second
ellipticity	1 minus the ratio of the 1-sigma lengths of the semimajor and semiminor axes
orientation	The angle (degrees) between the positive X axis and the major axis (increases counter-clockwise)
sky_orientation	The position angle (degrees) from North of the major axis
sky_bbox_ll	Sky coordinate of the lower-left vertex of the minimal bounding box of the source
sky_bbox_ul	Sky coordinate of the upper-left vertex of the minimal bounding box of the source
sky_bbox_lr	Sky coordinate of the lower-right vertex of the minimal bounding box of the source
sky_bbox_ur	Sky coordinate of the upper-right vertex of the minimal bounding box of the source

Note that pixel coordinates are 0 indexed, matching the Python 0-based indexing. That means pixel coordinate `0` is the center of the first pixel.

Segmentation Map

The segmentation map computed during the source finding process is saved to a single 2D image extension in a FITS file. Each image pixel contains an integer value corresponding to a source label number in the source catalog product. Pixels that don't belong to any source have a value of zero.

Arguments

The `source_catalog` step uses the following user-settable arguments:

- `--bkg_boxsize`: An integer value giving the background mesh box size in pixels
- `--kernel_fwhm`: A floating-point value giving the Gaussian kernel FWHM in pixels
- `--snr_threshold`: A floating-point value that sets the SNR threshold (above background) for source detection
- `--npixels`: An integer value that sets the minimum number of pixels in a source
- `--deblend`: A boolean indicating whether to deblend sources
- `--aperture_ee1`: An integer value of the smallest aperture encircled energy value
- `--aperture_ee2`: An integer value of the middle aperture encircled energy value
- `--aperture_ee3`: An integer value of the largest aperture encircled energy value
- `--ci1_star_threshold`: A floating-point value of the threshold compared to the concentration index calculated from aperture_ee1 and aperture_ee2 that is used to determine whether a source is a star. Sources must meet the criteria of both ci1_star_threshold and ci2_star_threshold to be considered a star.
- `--ci2_star_threshold`: A floating-point value of the threshold compared to the concentration index calculated from aperture_ee2 and aperture_ee3 that is used to determine whether a source is a star. Sources must meet the criteria of both ci1_star_threshold and ci2_star_threshold to be considered a star.
- `--suffix`: A string value giving the file name suffix to use for the output catalog file [default='cat']

Reference File Types

The `source_catalog` step uses APCORR, ABVEGAOFFSET, and PARS-SOURCECATALOGSTEP reference files.

APCORR Reference File

REFTYPE
APCORR

The APCORR reference file contains data necessary for correcting extracted imaging and spectroscopic photometry to the equivalent of an infinite aperture. It is used within the *source_catalog* step for imaging and within the *extract_1d* step for spectroscopic data.

Reference Selection Keywords for APCORR

CRDS selects appropriate APCORR references based on the following keywords. APCORR is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, EXP_TYPE, FILTER, GRATING, LAMP, OPMODE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for APCORR

In addition to the standard reference file keywords listed above, the following keywords are *required* in APCORR reference files, because they are used as CRDS selectors (see apcorr_selectors):

Keyword	Data Model Name	Instruments
EXP_TYPE	model.meta.exposure.type	All

NON-IFU APCORR Reference File Format

APCORR reference files for non-IFU data are in FITS format. The FITS APCORR reference file contains tabular data in a BINTABLE extension with EXTNAME = 'APCORR'. The FITS primary HDU does not contain a data array. The contents of the table extension varies for different instrument modes, as shown in the tables below.

Data model

`FgsImgApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.FgsImgApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
FGS	Image	eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

`MirImgApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirImgApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	Image	filter	string	12	N/A
		subarray	string	15	N/A
		eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

`MirLrsApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirLrsApcorrModel.html#jwst.dat

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	LRS	subarray	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	1D array	pixels
		nelem_size	integer	scalar	N/A
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model

`NrcImgApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrcImgApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRCam	Image	filter	string	12	N/A
		pupil	string	15	N/A
		eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

NrcWfssApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrcWfssApcorrModel.html#jwst.

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRCam	WFSS	filter	string	12	N/A
		pupil	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	1D array	pixels
		nelem_size	integer	scalar	N/A
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model

NisImgApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NisImgApcorrModel.html#jwst.dat

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRISS	Image	filter	string	12	N/A
		pupil	string	15	N/A
		eefraction	float	scalar	unitless
		radius	float	scalar	pixels
		apcorr	float	scalar	unitless
		skyin	float	scalar	pixels
		skyout	float	scalar	pixels

Data model

NisWfssApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NisWfssApcorrModel.html#jwst.

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRISS	WFSS	filter	string	12	N/A
		pupil	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	1D array	pixels
		nelem_size	integer	scalar	N/A
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model

NrsFsApcorrModel (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrsFsApcorrModel.html#jwst.datam

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	FS	filter	string	12	N/A
		grating	string	15	N/A
		slit	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	2D array	arcsec
		nelem_size	integer	scalar	N/A
		pixphase	float	1D array	N/A
		apcorr	float	3D array	unitless
		apcorr_err	float	3D array	unitless

Data model

`NrsMosApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.NrsMosApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	MOS	filter	string	12	N/A
		grating	string	15	N/A
		wavelength	float	1D array	micron
		nelem_wl	integer	scalar	N/A
		size	integer	2D array	arcsec
		nelem_size	integer	scalar	N/A
		pixphase	float	1D array	N/A
		apcorr	float	3D array	unitless
		apcorr_err	float	3D array	unitless

Row Selection

A row of data within the reference table is selected by the pipeline step based on the optical elements in use for the exposure. The selection attributes are always contained in the first few columns of the table. The remaining columns contain the data needed for the aperture correction. The row selection criteria for each instrument+mode are:

- **FGS Image:**
  - None (table contains a single row)
- **MIRI:**
  - Image: Filter and Subarray
  - LRS: Subarray
- **NIRCam:**
  - Image: Filter and Pupil
  - WFSS: Filter and Pupil
- **NIRISS:**
  - Image: Filter and Pupil
  - WFSS: Filter and Pupil
- **NIRSpec:**
  - MOS: Filter and Grating

- Fixed Slits: Filter, Grating, and Slit name

Note: Spectroscopic mode reference files contain the "nelem_wl" and "nelem_size" entries, which indicate to the pipeline step how many valid elements are contained in the "wavelength" and "size" arrays, respectively. Only the first "nelem_wl" and "nelem_size" entries are read from each array.

IFU APCORR Reference File ASDF Format

For IFU data the APCORR reference files are in ASDF format. The aperture correction varies with wavelength and the contents of the files are shown below. The radius, aperture correction and error are all 2D arrays. Currently the 2nd dimension does not add information, but in the future it could be used to provide different aperture corrections for different radii.

Data model
`MirMrsApcorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirMrsApcorrModel.html#jwst.da

Instrument	Mode	Column name	Data type	Dimensions	Units
MIRI	MRS	wavelength	float	1D array	micron
		radius	float	2D array	arcsec
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

Data model
`NRSIFUApcorrModel`

Instrument	Mode	Column name	Data type	Dimensions	Units
NIRSpec	MOS	filter	string	12	N/A
		grating	string	15	N/A
		wavelength	float	1D array	micron
		radius	float	2D array	arcsec
		apcorr	float	2D array	unitless
		apcorr_err	float	2D array	unitless

ABVEGAOFFSET Reference File

REFTYPE
ABVEGAOFFSET

Data model
`ABVegaOffsetModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ABVegaOffsetModel.html#jwst.dat

The ABVEGAOFFSET reference file contains data necessary for converting from AB to Vega magnitudes.

Reference Selection Keywords for ABVEGAOFFSET

CRDS selects appropriate ABVEGAOFFSET references based on the following keywords. ABVEGAOFFSET is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

ABVEGAOFFSET Reference File Format

ABVEGAOFFSET reference files are in ASDF format. The ABVEGAOFFSET reference file contains tabular data in a key called `abvega_offset`. The content of the table varies for different instrument modes, as shown in the tables below.

Instrument	Column name	Data type	Dimensions	Units
FGS	detector	string	7	N/A
	abvega_offset	float	scalar	unitless

Instrument	Column name	Data type	Dimensions	Units
MIRI	filter	string	12	N/A
	abvega_offset	float	scalar	unitless

Instrument	Column name	Data type	Dimensions	Units
NIRCam or NIRISS	filter	string	12	N/A
	pupil	string	15	N/A
	abvega_offset	float	scalar	unitless

Row Selection

A row of data within the reference table is selected by the pipeline step based on the optical elements in use for the exposure. The selection attributes are always contained in the first few columns of the table. The last column contains the data needed to convert from AB to Vega magnitudes. The row selection criteria for each instrument/mode are:

- **FGS:**
  - Detector
- **MIRI:**
  - Filter
- **NIRCam:**
  - Filter and Pupil
- **NIRISS:**
  - Filter and Pupil

PARS-SOURCECATALOGSTEP Parameter Reference File

REFTYPE
PARS-SOURCECATALOGSTEP

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-sourcecatalogstep references based on the following keywords.

Instrument	Keywords
FGS	EXP_TYPE
MIRI	EXP_TYPE, FILTER
NIRCAM	EXP_TYPE, FILTER
NIRISS	EXP_TYPE, FILTER

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

jwst.source_catalog Package

Classes

SourceCatalogStep([name, parent, ...])	Create a final catalog of source photometry and morphologies.

SourceCatalogStep

class jwst.source_catalog.**SourceCatalogStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

Create a final catalog of source photometry and morphologies.

Parameters

input (str or `ImageModel`) – A FITS filename or an `ImageModel` of a drizzled image.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.

- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input_model)	This is where real work happens.

Attributes Documentation

class_alias = 'source_catalog'

reference_file_types = ['apcorr', 'abvegaoffset']

spec

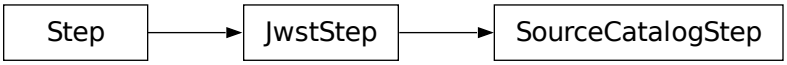
```
bkg_boxsize = integer(default=1000)   # background mesh box size in pixels
kernel_fwhm = float(default=2.0)      # Gaussian kernel FWHM in pixels
snr_threshold = float(default=3.0)    # SNR threshold above the bkg
npixels = integer(default=25)         # min number of pixels in source
deblend = boolean(default=False)      # deblend sources?
aperture_ee1 = integer(default=30)    # aperture encircled energy 1
aperture_ee2 = integer(default=50)    # aperture encircled energy 2
aperture_ee3 = integer(default=70)    # aperture encircled energy 3
ci1_star_threshold = float(default=2.0)  # CI 1 star threshold
ci2_star_threshold = float(default=1.8)  # CI 2 star threshold
suffix = string(default='cat')        # Default suffix for output files
```

Methods Documentation

process(*input_model*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.56 Spectral Leak

Description

Class
`jwst.spectral_leak.SpectralLeakStep`

Alias
spectral_leak

The MIRI MRS filters are designed to keep out-of-band light from interfering with the desired first order wavelengths dispersed in a given band. However, around 12.2 μm (channel 3A) a few-percent spectral leak admits second-order light from 6 μm (channel 1B) into the bandpass. This results in spectra produced by the pipeline containing additional flux around 12.2 μm that is only proportional to the object flux at 6 μm.

Applying the optimal spectral leak correction to MIRI MRS data in the *calwebb_spec3* pipeline corrects for this feature in extracted channel 3A spectrum for a given target using the channel 1B spectrum of that target (if available). Note that since the channel 1B FOV is smaller than that for Ch3A no such correction is possible in general for extended sources that fill the entire FOV. An example of an uncorrected and corrected spectrum is given in the figure below for a G dwarf star.

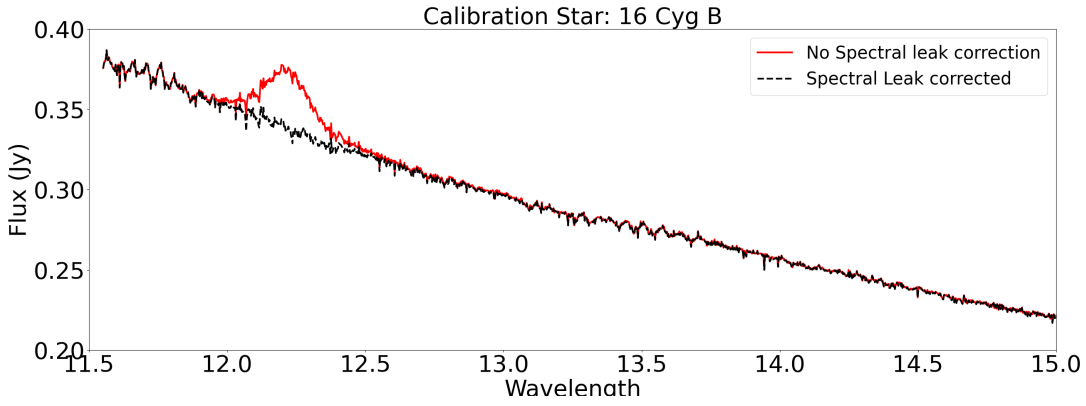


Figure: MRS spectral leak as seen in G Dwarf star. The red extracted spectrum does not have the spectral_leak step applied, while the the black extracted spectrum has the spectral leak correction applied.

Step Arguments

The spectral_leak correction has no step-specific arguments.

Reference Files

The `spectral_leak` step uses the MRSPTCORR reference file.

MRSPTCORR reference file

REFTYPE
MRSPTCORR

Data models
`MirMrsPtCorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirMrsPtCorrModel.html#jwst.dat

The MRSPTCORR reference file contains parameter values used to subtract the MRS 12 micron spectral leak in the spectral leak step. It also contains parameters to correct point sources, in future enhancements, for across-slice corrections and throughput variations.

Reference Selection Keywords for MRSPTCORR

CRDS selects appropriate MRSPTCORR references based on the following keywords. MRSPTCORR is not applicable for instruments not in the table.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for MRSPTCORR

In addition to the standard reference file keywords listed above, the following keywords are *required* in MRSPTCORR reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for MRSPTCORR*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

MIRI Reference File Format

The MIRI MRSPTCORR reference files are FITS format, with 5 BINTABLE extensions. The FITS primary data array is assumed to be empty. The format and content of the MIRI MRSPTCORR reference file

EXTNAME	XTENSION	Dimensions
LEAKCOR	BINTABLE	TFIELDS = 3
TRACOR	BINTABLE	TFIELDS = 7
WAVCORR_OPTICAL	BINTABLE	TFIELDS = 6
WAVCORR_XSLICE	BINTABLE	TFIELDS = 2
WAVCORR_SHIFT	BINTABLE	TFIELDS = 3

The formats of the individual table extensions are listed below.

Table	Column	Data type	Units
LEAKCOR	WAVELENGTH	float	micron
	FRAC_LEAK	float	N/A
	ERR_LEAK	float	N/A
TRACOR	CHANNEL	int	N/A
	WAVE_MIN	float	micron
	WAVE_MAX	float	micron
	T_WMIN_CENTRE	float	percent
	T_WMIN_EDGE	float	percent
	T_WMAX_CENTRE	float	percent
	T_WMAX_EDGE	float	percent
WAVCORR_OPTICAL	SUB_BAND	string	N/A
	BETA_SLICE	float	arcsec
	WAVE_MIN	float	micron
	WAVE_MAX	float	micron
	SRP_MIN	float	N/A
	SRP_MAX	float	N/A
WAVCORR_XSLICE	XSLICE_MIN	float	micron/arcsec
	XSLICE_MAX	float	micron/arcsec
WAVCORR_SHIFT	BETA_OFF	float	slice
	DS_MIN	float	slice
	DS_MAX	float	slice

This reference file contains the relative spectrophotometric response function for the MRS 12 micron spectral leak, along with tables of across-slice wavelength and throughput variations for point sources in each of the MRS bands. Currently, only the LEAKCOR table is use in the jwst pipeline.

jwst.spectral_leak.spectral_leak_step Module

Classes

SpectralLeakStep([name, parent, ...])	The MIRI MRS has a spectral leak in which 6 micron light leaks into the 12 micron channel.

SpectralLeakStep

class jwst.spectral_leak.spectral_leak_step.**SpectralLeakStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

The MIRI MRS has a spectral leak in which 6 micron light leaks into the 12 micron channel. This step applies a correction to the 12 micron channel.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types

Methods Summary

process(input)	Execute the step.

Attributes Documentation

class_alias = 'spectral_leak'

reference_file_types = ['mrsptcorr']

Methods Documentation

process(*input*)

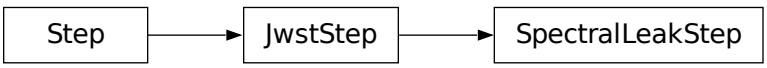
Execute the step.

Parameters
input (*container of models containing 1-D extracted spectra*) –

Returns
The corrected data model. This will be "input" if the step is skipped, otherwise it will be a corrected 1D extracted spectrum that contains the MRS channel 3B range.

Return type
JWST DataModel

Class Inheritance Diagram



15.1.57 Source Type (SRCTYPE) Determination

Description

Class
jwst.srctype.SourceTypeStep

Alias
srctype

The Source Type (`srctype`) step in the calibration pipeline attempts to determine whether a spectroscopic source should be considered to be a point or extended object, populating the "SRCTYPE" keyword with a value of either "POINT" or "EXTENDED." This information is then used in some subsequent spectroscopic processing steps to apply source-dependent corrections.

Single Source Observations

For JWST observing modes that use a single primary target (e.g. MIRI MRS and LRS spectroscopy and NIRSpec IFU and Fixed-Slit spectroscopy), the observer has the option of designating a source type in the APT template for the observation. They have the choice of declaring whether or not the source should be considered extended. If they don't know the character of the source, they can also choose a value of "UNKNOWN." The observer's choice is passed along to DMS processing, which sets the value of the "SRCTYAPT" keyword in the primary header of the products used as input to the calibration pipeline. If the user has selected a value in the APT, the "SRCTYAPT" keyword will be set to "POINT", "EXTENDED", or "UNKNOWN." If the selection is not available for a given observing mode or a choice wasn't made, the "SRCTYAPT" keyword will not appear in the uncalibrated product header.

The `srctype` step sets a value for the "SRCTYPE" keyword that is stored in the "SCI" extension header(s) of data products. The step sets the value of "SRCTYPE" based on input from the user given in the "SRCTYAPT" keyword, as well as other rules that can override the "SRCTYAPT" values.

The `srctype` step first checks to see if the "SRCTYAPT" keyword is present and has already been populated. If "SRCTYAPT" is not present or is set to "UNKNOWN", the step determines a suitable value based on the observing mode, command line input, and other characteristics of the exposure. The following choices are used, in order of priority:

1. The source type can be specified by the user on the command line. Exposure types for which this is permitted contain a single pre-defined target, i.e. MIR_LRS-FIXEDSLIT, MIR_LRS-SLITLESS, MIR_MRS,NRC_TSGRISM, NRS_FIXEDSLIT, NRS_BRIGHTOBJ, and NRS_IFU. Other EXP_TYPEs will be ignored. For NRS_FIXEDSLIT exposures, a user-supplied value can replace the value for the target in the primary slit only, while the other slits will retain their default settings of "EXTENDED" (which is appropriate for sky background).
2. Background target exposures default to a source type of "EXTENDED." Background exposures are identified by the keyword "BKGDTARG" set to True.
3. TSO exposures default to a source type of "POINT." TSO exposures are identified by EXP_TYPE="NRC_TSGRISM" or "NRS_BRIGHTOBJ", or TSOVISIT=True.
4. Exposures that are part of a nodded dither pattern, which are assumed to only be used with point-like targets, default to a source type of "POINT." Nodded exposures are usually identified by the "PATTTYPE" keyword either being set to a value of "POINT-SOURCE" or containing the sub-string "NOD" (NIRSpec IFU and Fixed Slit). For MIRI MRS exposures the keyword "DITHOPFR" (DITHer pattern OPtimized FoR) is used instead of "PATTTYPE". If it has a value of "POINT-SOURCE", the source type is set to "POINT".
5. If none of the above conditions apply, and the user did not choose a value in the APT, the following table of defaults is used, based on the "EXP_TYPE" keyword value:

EXP_TYPE	Exposure Type	SRCTYPE
MIR_LRS-FIXEDSLIT	MIRI LRS fixed-slit	POINT
MIR_LRS-SLITLESS	MIRI LRS slitless	POINT
MIR_MRS	MIRI MRS (IFU)	EXTENDED
NIS_SOSS	NIRISS SOSS	POINT
NRS_FIXEDSLIT	NIRSpec fixed-slit	POINT
NRS_BRIGHTOBJ	NIRSpec bright object	POINT
NRS_IFU	NIRSpec IFU	EXTENDED

If the EXP_TYPE value of the input image is not in the above list, SRCTYPE will be set to "UNKNOWN".

NOTE: NIRSpec fixed-slit (EXP_TYPE="NRS_FIXEDSLIT") exposures are unique in that a single target is specified in the APT, yet data for multiple slits can be contained within an exposure, depending on the size of the readout used (e.g. SUBARRAY="ALLSLITS"). For this observing mode, the source type selection resulting from the logic

outlined above is used to populate the SRCTYPE keyword associated with the data for the primary slit instance in the pipeline data products. The primary slit is determined from the value of the "FXD_SLIT" keyword. Any additional slit instances contained within the data product will have their SRCTYPE value set to "EXTENDED", as non-primary slits are expected to contain background.

Multi-Source Observations

NIRSpec MOS

For NIRSpec MOS exposures (EXP_TYPE="NRS_MSASPEC"), there are multiple sources per exposure and hence a single user-selected parameter can't be used in the APT, nor a single keyword in the science product, to record the type of each source. For these exposures, a stellarity value can be supplied by the observer for each source used in the MSA Planning Tool (MPT). The stellarity values are in turn passed from the MPT to the MSA metadata (_msa.fits) file created by DMS and used in the calibration pipeline. The stellarity values from the MSA metadata file are loaded for each source/slitlet by the `assign_wcs` step of the *calwebb_spec2* pipeline and then evaluated by the `srctype` step to determine whether each source should be treated as point or extended.

If the stellarity value for a given source in the MSA metadata is less than zero, the source type defaults to "POINT." If the stellarity value is between zero and 0.75, it is set to "EXTENDED", and if the stellarity value is greater than 0.75, it is set to "POINT." The resulting choice is stored in the "SRCTYPE" keyword located in the header of the SCI extension associated with each slitlet.

In the future, reference files will be used to set more detailed threshold values for stellarity, based on the particular filters, gratings, etc. of each exposure.

NIRCam and NIRISS WFSS

It is not possible to specify ahead of time the source types for spectra that may show up in a Wide-Field Slitless Spectroscopy exposure. So for these modes the `srctype` step uses the value from the `is_extended` column of the source catalog generated from the direct imaging taken with WFSS observations and uses that to set "POINT" or "EXTENDED" for each extracted source.

Step Arguments

The Source Type step uses the following optional argument.

`--source_type`
: A string that can be used to override the source_type that will be written to the SRCTYPE keyword. The allowed values are `POINT` and `EXTENDED`.

Reference File

The Source Type step does not use any reference files.

jwst.srctype Package

Classes

SourceTypeStep([name, parent, config_file, ...])	SourceTypeStep: Selects and sets a source type based on various inputs.

SourceTypeStep

class jwst.srctype.**SourceTypeStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

SourceTypeStep: Selects and sets a source type based on various inputs. The source type is used in later calibrations to determine the appropriate methods to use. Input comes from either the SRCTYAPT keyword value, which is populated from user info in the APT, or the NIRSpec MSA planning tool. The source type can be also specified on the command line for exposures containing a single pre-defined target.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'srctype'

spec

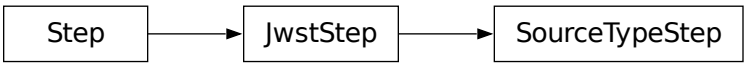
```
source_type = option('POINT','EXTENDED', default=None)  # user-supplied source
↪type
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.58 Stack PSF References

Description

Class
`jwst.coron.StackRefsStep`

Alias
stack_refs

The `stack_refs` step is one of the coronagraphic-specific steps in the `coron` sub-package and is part of Stage 3 *calwebb_coron3* processing. It takes a list of reference PSF products and stacks all of the per-integration images contained in each PSF product into a single 3D data cube. This operation prepares the PSF images for use by subsequent steps in the *calwebb_coron3* pipeline. The image data are simply copied and reformatted, without being modified in any way.

Arguments

The `stack_refs` step does not have any step-specific arguments.

Inputs

3D calibrated images

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_calints

The inputs to the `stack_refs` step are multiple calibrated products for the PSF target, produced by the *calwebb_image2* pipeline. Each input should be a 3D "_calints" product, containing a 3D stack of calibrated images for the multiple integrations within each exposure.

It is assumed that the `stack_refs` step will be called from the *calwebb_coron3* pipeline, which is given an ASN file as input, specifying one or more PSF target exposures. The actual input passed to the `stack_refs` step will be a `ModelContainer` created by the *calwebb_coron3* pipeline, containing a `CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeModel) data model for each PSF "_calints" exposure listed in the ASN file. See *calwebb_coron3* for more details on the contents of the ASN file.

Outputs

3D PSF image stack

Data model
`CubeModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.CubeModel.html#jwst.datamodels.CubeMode

File suffix
_psfstack

The output of the `stack_refs` step will be a single 3D product containing a stack of all the PSF images from the multiple input exposures. The size of the stack will be equal to the sum of the number of integration (NINTS) in each input PSF exposure. The output file name is source-based, using the product name specified in the ASN file, e.g. "jw86073-a3001_t001_nircam_f140m-maskbar_psfstack.fits."

Reference Files

The `stack_refs` step does not use any reference files.

jwst.coron.stack_refs_step Module

Classes

StackRefsStep([name, parent, config_file, ...])	StackRefsStep: Stack multiple PSF reference exposures into a single CubeModel, for use by subsequent coronagraphic steps.

StackRefsStep

class jwst.coron.stack_refs_step.**StackRefsStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

StackRefsStep: Stack multiple PSF reference exposures into a single CubeModel, for use by subsequent coronagraphic steps.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'stack_refs'

spec

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.59 STPIPE

For Users

Steps

Configuring a Step

This section describes how to instantiate a Step and set configuration parameters on it.

Steps can be configured by either:

- Writing a parameter file
- Instantiating the Step directly from Python

Running a Step from a parameter file

A parameter file contains one or more of a `Step`'s parameters. Any parameter not specified in the file will take its value from the CRDS-retrieved parameter reference file or the defaults coded directly into the `Step`. Note that any parameter specified on the command line overrides all other values.

The preferred format of parameter files is the *ASDF Parameter Files* format. Refer to the *minimal example* for a complete description of the contents. The rest of this document will focus on the step parameters themselves.

Every parameter file must contain the key `class`, followed by the optional `name` followed by any parameters that are specific to the step being run.

`class` specifies the Python class to run. It should be a fully-qualified Python path to the class. Step classes can ship with `stpipe` itself, they may be part of other Python packages, or they exist in freestanding modules alongside the configuration file. For example, to use the `SystemCall` step included with `stpipe`, set `class` to `stpipe.subprocess.SystemCall`. To use a class called `Custom` defined in a file `mysteps.py` in the same directory as the configuration file, set `class` to `mysteps.Custom`.

`name` defines the name of the step. This is distinct from the class of the step, since the same class of Step may be configured in different ways, and it is useful to be able to have a way of distinguishing between them. For example, when Steps are combined into *Pipelines*, a Pipeline may use the same Step class multiple times, each with different configuration parameters.

The parameters specific to the Step all reside under the key `parameters`. The set of accepted parameters is defined in the Step's spec member. The easiest way to get started on a parameter file is to call `Step.export_config` and then edit the file that is created. This will generate an ASDF config file that includes every available parameter, which can then be trimmed to the parameters that require customization.

Here is an example parameter file (`do_cleanup.asdf`) that runs the (imaginary) step `stpipe.cleanup` to clean up an image.

```
#ASDF 1.0.0
#ASDF_STANDARD 1.3.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
class: stpipe.cleanup
name: MyCleanup
parameters:
  threshold: 42.0
  scale: 0.01
...
```

Running a Step from the commandline

The `strun` command can be used to run Steps from the commandline.

The first argument may be either:

- The path to a parameter file
- A Python class

Additional parameters may be passed on the commandline. These parameters override any that are present in the parameter file. Any extra positional parameters on the commandline are passed to the step's process method. This will often be input filenames.

For example, to use an existing parameter file from above, but override it so the threshold parameter is different:

```
$ strun do_cleanup.asdf input.fits --threshold=86
```

To display a list of the parameters that are accepted for a given Step class, pass the `-h` parameter, and the name of a Step class or parameter file:

```
$ strun -h do_cleanup.asdf
usage: strun [--logcfg LOGCFG] cfg_file_or_class [-h] [--pre_hooks]
             [--post_hooks] [--skip] [--scale] [--extname]
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help       show this help message and exit
  --logcfg LOGCFG  The logging configuration file to load
  --verbose, -v    Turn on all logging messages
  --debug          When an exception occurs, invoke the Python debugger, pdb
  --pre_hooks
  --post_hooks
  --skip           Skip this step
  --scale          A scale factor
  --threshold      The threshold below which to apply cleanup
  --output_file    File to save the output to
```

Every step has an `--output_file` parameter. If one is not provided, the output filename is determined based on the input file by appending the name of the step. For example, in this case, `foo.fits` is output to `foo_cleanup.fits`.

Finally, the parameters a `Step` actually ran with can be saved to a new parameter file using the `--save-parameters` option. This file will have all the parameters, specific to the step, and the final values used.

Parameter Precedence

There are a number of places where the value of a parameter can be specified. The order of precedence, from most to least significant, for parameter value assignment is as follows:

1. Value specified on the command-line: `strun step.asdf --par=value_that_will_be_used`
2. Value found in the user-specified parameter file
3. CRDS-retrieved parameter reference
4. `Step`-coded default, determined by the parameter definition `Step.spec`

For pipelines, if a pipeline parameter file specifies a value for a step in the pipeline, that takes precedence over any step-specific value found, either from a step-specific parameter file or CRDS-retrieved step-specific parameter file. The full order of precedence for a pipeline and its sub steps is as follows:

1. Value specified on the command-line: `strun pipeline.asdf --steps.step.par=value_that_will_be_used`
2. Value found in the user-specified pipeline parameter file: `strun pipeline.asdf`
3. Value found in the parameter file specified in a pipeline parameter file
4. CRDS-retrieved parameter reference for the pipeline
5. CRDS-retrieved parameter reference for each sub-step
6. `Pipeline`-coded default for itself and all sub-steps
7. `Step`-coded default for each sub-step

Debugging

To output all logging output from the step, add the `--verbose` option to the commandline. (If more fine-grained control over logging is required, see *Logging*).

To start the Python debugger if the step itself raises an exception, pass the `--debug` option to the commandline.

CRDS Retrieval of Step Parameters

In general, CRDS uses the input to a `Step` to determine which reference files to use. Nearly all JWST-related steps take only a single input file. However, often times that input file is an association. Since step parameters are configured only once per execution of a step or pipeline, only the first qualifying member, usually of type `science` is used.

Retrieval of `Step` parameters from CRDS can be completely disabled by using the `--disable-crds-steppars` command-line switch, or setting the environment variable `STPIPE_DISABLE_CRDS_STEPPARS` to `true`.

Running a Step in Python

There are a number of methods to run a step within a Python interpreter, depending on how much control one needs.

Step.from_cmdline()

For individuals who are used to using the `strun` command, `Step.from_cmdline` is the most direct method of executing a step or pipeline. The only argument is a list of strings, representing the command line arguments one would have used for `strun`. The call signature is:

```
Step.from_cmdline([string,...])
```

For example, given the following command-line:

```
$ strun calwebb_detector1 jw00017001001_01101_00001_nrca1_uncal.fits
       --steps.linearity.override_linearity='my_lin.fits'
```

the equivalent `from_cmdline` call would be:

```
from jwst.pipeline import Detector1Pipeline
Detector1Pipeline.from_cmdline(['jw00017001001_01101_00001_nrca1_uncal.fits',
                                'steps.linearity.override_linearity', 'my_lin.fits'])
```

call()

Class method `Step.call` is the slightly more programmatic, and preferred, method of executing a step or pipeline. When using `call`, one gets the full configuration initialization, including CRDS parameter reference retrieval, that one gets with the `strun` command or `Step.from_cmdline` method. The call signature is:

```
Step.call(input, config_file=None, **parameters)
```

The positional argument `input` is the data to be operated on, usually a string representing a file path or a *DataModel* The optional keyword argument `config_file` is used to specify a local parameter file. The optional keyword argument `logcfg` is used to specify a logging configuration file. Finally, the remaining optional keyword arguments are the parameters that the particular step accepts. The method returns the result of the step. A basic example is:

```
from jwst.jump import JumpStep
output = JumpStep.call('jw00017001001_01101_00001_nrca1_uncal.fits')
```

makes a new instance of `JumpStep` and executes using the specified exposure file. `JumpStep` has a parameter `rejection_threshold`. To use a different value than the default, the statement would be:

```
output = JumpStep.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                       rejection_threshold=42.0)
```

If one wishes to use a *parameter file*, specify the path to it using the `config_file` argument:

```
output = JumpStep.call('jw00017001001_01101_00001_nrca1_uncal.fits',
                       config_file='my_jumpstep_config.asdf')
```

run()

The instance method `Step.run()` is the lowest-level method to executing a step or pipeline. Initialization and parameter settings are left up to the user. An example is:

```
from jwst.flatfield import FlatFieldStep

mystep = FlatFieldStep()
mystep.override_sflat = 'sflat.fits'
output = mystep.run(input)
```

`input` (https://docs.python.org/3/library/functions.html#input) in this case can be a fits file containing the appropriate data, or the output of a previously run step/pipeline, which is an instance of a particular *datamodel*.

Unlike the `call` class method, there is no parameter initialization that occurs, either by a local parameter file or from a CRDS-retrieved parameter reference file. Parameters can be set individually on the instance, as is shown above. Parameters can also be specified as keyword arguments when instantiating the step. The previous example could be re-written as:

```
from jwst.flatfield import FlatFieldStep

mystep = FlatFieldStep(override_sflat='sflat.fits')
output = mystep.run(input)
```

One can implement parameter reference file retrieval and use of a local parameter file as follows:

```
from stpipe import config_parser
from jwst.flatfield import FlatFieldStep

config = FlatFieldStep.get_config_from_reference(input)
local_config = config_parser.load_config_file('my_flatfield_config.asdf')
config_parser.merge_config(config, local_config)

flat_field_step = FlatFieldStep.from_config_section(config)
output = flat_field_step.run(input)
```

Using the `.run()` method is the same as calling the instance directly. They are equivalent:

```
output = mystep(input)
```

Pipelines

It is important to note that a Pipeline is also a Step, so everything that applies to a Step in the *For Users* chapter also applies to Pipelines.

Configuring a Pipeline

This section describes how to set parameters on the individual steps in a pipeline. To change the order of steps in a pipeline, one must write a Pipeline subclass in Python. That is described in the *Pipelines* section of the developer documentation.

Just as with Steps, Pipelines can by configured either by a parameter file or directly from Python.

From a parameter file

A Pipeline parameter file follows the same format as a Step parameter file: *ASDF Parameter Files*

Here is an example pipeline parameter file for the `Image2Pipeline` class:

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: Space Telescope Science Institute, homepage:
↪'http://github.com/spacetelescope/asdf',
   name: asdf, version: 2.7.3}
class: jwst.pipeline.Image2Pipeline
name: Image2Pipeline
parameters:
   save_bsub: false
steps:
- class: jwst.flatfield.flat_field_step.FlatFieldStep
  name: flat_field
  parameters:
    skip = True
- class: jwst.resample.resample_step.ResampleStep
  name: resample
  parameters:
    pixel_scale_ratio: 1.0
    pixfrac: 1.0
```

Just like a `Step`, it must have `name` and `class` values. Here the `class` must refer to a subclass of `stpipe.Pipeline`.

Following `name` and `class` is the `steps` section. Under this section is a subsection for each step in the pipeline. The easiest way to get started on a parameter file is to call `Step.export_config` and then edit the file that is created. This will generate an ASDF config file that includes every available parameter, which can then be trimmed to the parameters that require customization.

For each Step's section, the parameters for that step may either be specified inline, or specified by referencing an external parameter file just for that step. For example, a pipeline parameter file that contains:

```
steps:
- class: jwst.resample.resample_step.ResampleStep
  name: resample
  parameters:
    pixel_scale_ratio: 1.0
    pixfrac: 1.0
```

is equivalent to:

```
steps:
- class: jwst.resample.resample_step.ResampleStep
  name: resample
  parameters:
     config_file = myresample.asdf
```

with the file `myresample.asdf.` in the same directory:

```
class: jwst.resample.resample_step.ResampleStep
name: resample
parameters:
  pixel_scale_ratio: 1.0
  pixfrac: 1.0
```

If both a `config_file` and additional parameters are specified, the `config_file` is loaded, and then the local parameters override them.

Any optional parameters for each Step may be omitted, in which case defaults will be used.

From Python

A pipeline may be configured from Python by passing a nested dictionary of parameters to the Pipeline's constructor. Each key is the name of a step, and the value is another dictionary containing parameters for that step. For example, the following is the equivalent of the parameter file above:

```
from stpipe.pipeline import Image2Pipeline

steps = {
    'resample': {'pixel_scale_ratio': 1.0, 'pixfrac': 1.0}
}

pipe = Image2Pipeline(steps=steps)
```

Running a Pipeline

From the commandline

The same `strun` script used to run Steps from the commandline can also run Pipelines.

The only wrinkle is that any parameters overridden from the commandline use dot notation to specify the parameter name. For example, to override the `pixfrac` value on the `resample` step in the example above, one can do:

```
> strun stpipe.pipeline.Image2Pipeline --steps.resample.pixfrac=2.0
```

From Python

Once the pipeline has been configured (as above), just call the instance to run it.

```
pipe()
```

Caching details

The results of a Step are cached using Python pickles. This allows virtually most of the standard Python data types to be cached. In addition, any FITS models that are the result of a step are saved as standalone FITS files to make them more easily used by external tools. The filenames are based on the name of the substep within the pipeline.

Hooks

Each Step in a pipeline can also have pre- and post-hooks associated. Hooks themselves are Step instances, but there are some conveniences provided to make them easier to specify in a parameter file.

Pre-hooks are run right before the Step. The inputs to the pre-hook are the same as the inputs to their parent Step. Post-hooks are run right after the Step. The inputs to the post-hook are the return value(s) from the parent Step. The return values are always passed as a list. If the return value from the parent Step is a single item, a list of this single item is passed to the post hooks. This allows the post hooks to modify the return results, if necessary.

Hooks are specified using the `pre_hooks` and `post_hooks` parameters associated with each step. More than one pre- or post-hook may be assigned, and they are run in the order they are given. There can also be `pre_hooks` and `post_hooks` on the Pipeline as a whole (since a Pipeline is also a Step). Each of these parameters is a list of strings, where each entry is one of:

- An external commandline application. The arguments can be accessed using {0}, {1} etc. (See `stpipe.subproc.SystemCall`).
- A dot-separated path to a Python Step class.
- A dot-separated path to a Python function.

For example, here's a `post_hook` that will display a FITS file in the `ds9` FITS viewer the `flat_field` step has done flat field correction on it:

```
steps:
- class: jwst.resample.resample_step.ResampleStep
  name: resample
  parameters:
     post_hooks = "ds9 {0}",
```

Logging

Log messages are emitted from each Step at different levels of importance. The levels used are the standard ones for Python (from least important to most important:

1. DEBUG
2. INFO
3. WARNING
4. ERROR
5. CRITICAL

By default, only messages of type WARNING or higher are displayed. This can be controlled by providing a logging configuration file.

Logging configuration

A logging configuration file can be provided to customize what is logged.

A logging configuration file is searched for in the following places. The first one found is used *in its entirety* and all others are ignored:

1. The file specified with the `--logcfg` option to the `strun` script.
2. The file specified with the `logcfg` keyword to a .call() execution of a Step or Pipeline.
3. A file called `stpipe-log.cfg` in the current working directory.
4. `~/stpipe-log.cfg`
5. `/etc/stpipe-log.cfg`

The logging configuration file is in the standard ini-file format.

Each section name is a Unix-style filename glob pattern used to match a particular Step's logger. The settings in that section apply only to that Steps that match that pattern. For example, to have the settings apply to all steps, create a section called `[*]`. To have the settings apply only to a Step called `MyStep`, create a section called `[*.MyStep]`. To apply settings to all Steps that are substeps of a step called `MyStep`, call the section `[*.MyStep.*]`.

In each section, the following may be configured:

1. `level`: The level at and above which logging messages will be displayed. May be one of (from least important to most important): DEBUG, INFO, WARNING, ERROR or CRITICAL.
2. `break_level`: The level at and above which logging messages will cause an exception to be raised. For instance, if you would rather stop execution at the first ERROR message (rather than continue), set `break_level` to `ERROR`.
3. `handler`: Defines where log messages are to be sent. By default, they are sent to stderr. However, one may also specify:
   - `file:filename.log` to send the log messages to the given file.
   - `append:filename.log` to append the log messages to the given file. This is useful over `file` if multiple processes may need to write to the same log file.
   - `stdout` to send log messages to stdout.

   Multiple handlers may be specified by putting the whole value in quotes and separating the entries with a comma.

4. `format`: Allows one to customize what each log message contains. What this string may contain is described in the logging module LogRecord Attributes (https://docs.python.org/3/library/logging.html#logrecord-attributes) section of the Python standard library.

Examples

The following configuration turns on all log messages and saves them in the file myrun.log:

```
[*]
level = INFO
handler = file:myrun.log
```

In a scenario where the user is debugging a particular Step, they may want to hide all logging messages except for that Step, and stop when hitting any warning for that Step:

```
[*]
level = CRITICAL

[*.MyStep]
level = INFO
break_level = WARNING
```

ASDF Parameter Files

ASDF is the format of choice for parameter files. ASDF (https://asdf-standard.readthedocs.io/) stands for "Advanced Scientific Data Format", a general purpose, non-proprietary, and system-agnostic format for the dissemination of data. Built on YAML (https://yaml.org/), the most basic file is text-based requiring minimal formatting.

ASDF replaces the original *CFG* format for step configuration. Using ASDF allows the configurations to be stored and retrieved from CRDS, selecting the best parameter file for a given set of criteria, such as instrument and observation mode.

To create a parameter file, the most direct way is to choose the Pipeline class, Step class, or already existing .asdf or .cfg file, and run that step using the `--save-parameters` option. For example, to get the parameters for the `Spec2Pipeline` pipeline, do the following:

```
$ strun jwst.pipeline.Spec2Pipeline jw00017001001_01101_00001_nrs1_uncal.fits --save-
↪parameters my_spec2.asdf
```

Once created and modified as necessary, the file can now be used by `strun` to run the step/pipeline with the desired parameters:

```
$ strun my_spec2.asdf jw00017001001_01101_00001_nrs1_uncal.fits
```

The remaining sections will describe the file format and contents.

File Contents

To describe the contents of an ASDF file, the configuration for the step `CubeBuildStep` will be used as the example:

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: Space Telescope Science Institute, homepage:
↪'http://github.com/spacetelescope/asdf',
  name: asdf, version: 2.7.3}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.7.3}
class: jwst.cube_build.cube_build_step.CubeBuildStep
name: CubeBuildStep
parameters:
  band: all
  channel: all
  coord_system: skyalign
  filter: all
  grating: all
  input_dir: ''
  output_ext: .fits
  output_type: band
  output_use_index: true
  output_use_model: true
  post_hooks: []
  pre_hooks: []
  rois: 0.0
  roiw: 0.0
  save_results: false
  scale1: 0.0
  scale2: 0.0
  scalew: 0.0
  search_output_file: false
  single: false
  skip: false
  skip_dqflagging: false
  weight_power: 2.0
  weighting: emsm
  ...
```

Required Components

Preamble

The first 5 lines, up to and including the "—" line, define the file as an ASDF file. The rest of the file is formatted as one would format YAML data. Being YAML, the last line, containing the three `...` is essential.

class and name

There are two required keys at the top level: `class` and `parameters`. `parameters` is discussed below.

`class` specifies the Python class to run. It should be a fully-qualified Python path to the class. Step classes can ship with `stpipe` itself, they may be part of other Python packages, or they exist in freestanding modules alongside the configuration file. For example, to use the `SystemCall` step included with `stpipe`, set `class` to `stpipe.subprocess.SystemCall`. To use a class called `Custom` defined in a file `mysteps.py` in the same directory as the configuration file, set `class` to `mysteps.Custom`.

`name` defines the name of the step. This is distinct from the class of the step, since the same class of Step may be configured in different ways, and it is useful to be able to have a way of distinguishing between them. For example, when Steps are combined into *Pipelines*, a Pipeline may use the same Step class multiple times, each with different configuration parameters.

Parameters

`parameters` contains all the parameters to pass onto the step. The order of the parameters does not matter. It is not necessary to specify all parameters either. If not defined, the default, as defined in the code or values from CRDS parameter references, will be used.

Formatting

YAML has two ways of formatting a list of key/value pairs. In the above example, each key/value pair is on separate line. The other way is using a form that is similar to a Python `dict`. For example, the `parameters` block above could also have been formatted as:

```
parameters: {band: all, channel: all, coord_system: world, filter: all,
  grating: all, output_type: band, output_use_model: true, rois: 0.0,
  roiw: 0.0, scale1: 0.0, scale2: 0.0, scalew: 0.0, search_output_file: false,
  single: false, skip_dqflagging: false, weight_power: 2.0, weighting: msm}
```

Optional Components

The `asdf_library` and `history` blocks are necessary only when a parameter file is to be used as a parameter reference file in CRDS. See *Parameter Files as Reference Files* below.

Completeness

For any parameter file, it is not necessary to specify all step/pipeline parameters. Any parameter left unspecified will get, at least, the default value define in the step's code. If a parameter is defined without a default value, and the parameter is never assigned a value, an error will be produced when the step is executed.

Remember that parameter values can come from numerous sources. Refer to *Parameter Precedence* for a full listing of how parameters can be set.

From the `CubeBuildStep` example, if all that needed to change is the `weight_power` parameter with a setting of `4.0`, the `parameters` block need only contain the following:

```
parameters:
  weight_power: 4.0
```

Pipeline Configuration

Pipelines are essentially steps that refer to sub-steps. As in the original cfg format, parameters for sub-steps can also be specified. All sub-step parameters appear in a key called `steps`. Sub-step parameters are specified by using the sub-step name as the key, then underneath and indented, the parameters to change for that sub-step. For example, to define the `weight_power` of the `cube_build` step in a `Spec2Pipeline` parameter file, the parameter block would look as follows:

```
class: jwst.pipeline.Spec2Pipeline
parameters: {}
steps:
- class: jwst.cube_build.cube_build_step.CubeBuildStep
  parameters:
    weight_power: 4.0
```

As with step parameter files, not all sub-steps need to be specified. If left unspecified, the sub-steps will be run with their default parameter sets. For the example above, the other steps of `Spec2Pipeline`, such as `assign_wcs` and `photom` would still be executed.

Similarly, to skip a particular step, one would specify `skip: true` for that substep. Continuing from the above example, to skip the `msa_flagging` step, the parameter file would look like:

```
class: jwst.pipeline.Spec2Pipeline
parameters: {}
steps:
- class: jwst.msaflagopen.msaflagopen_step.MSAFlagOpenStep
  parameters:
    skip: true
- class: jwst.cube_build.cube_build_step.CubeBuildStep
  parameters:
    weight_power: 4.0
```

Note: In the previous examples, one may have noted the line `parameters: {}`. In neither example, and is a common situation when defining pipeline configurations, there is no need to set any of the parameters for the pipeline itself. However, the keyword `parameters` is required. As such, the value for `parameters` is defined as an empty dictionary, `{}`.

Python API

There are a number of ways to create an ASDF parameter file. From the command line utility `strun`, the option `--save-parameters` can be used.

Within a Python script, the method `Step.export_config(filename: str)` can be used. For example, to create a parameter file for `CubeBuildStep`, use the following:

```
>>> from jwst.cube_build import CubeBuildStep
>>> step = CubeBuildStep()
>>> step.export_config('cube_build.asdf')
```

Parameter Files as Reference Files

ASDF-formatted parameter files are the basis for the parameter reference reftypes in CRDS. There are two more keys that are needed to be added which CRDS requires: `meta` and `history`.

The direct way of creating a parameter reference file is through the `Step.export_config` method, just as one would to get a basic parameter file. The only addition is the argument `include_meta=True`. For example, to get a reference-file ready version of the `CubeBuildStep`, use the following Python code:

```
>>> from jwst.cube_build import CubeBuildStep
>>> step = CubeBuildStep()
>>> step.export_config('pars-cubebuildstep.asdf', include_meta=True)
```

The explanations for the `meta` and `history` blocks are given below.

META Block

When a parameter file is to be ingested into CRDS, there is another key required, `meta`, which defines the information needed by CRDS parameter file selection. A basic reference parameter file will look as follows:

```
#ASDF 1.0.0
#ASDF_STANDARD 1.3.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
history:
  entries:
  - !core/history_entry-1.0.0 {description: Base values, time: !!timestamp '2019-10-29
      21:20:50'}
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: {name: asdf, version: 2.4.2}
meta:
   author: Alfred E. Neuman
   date: '2019-07-17T10:56:23.456'
   description: MakeListStep parameters
   instrument: {name: MIRI}
   pedigree: GROUND
   reftype: pars-spec2pipeline
```

(continues on next page)

(continued from previous page)

```
   telescope: JWST
   title: Spec2Pipeline default parameters
   useafter: '1990-04-24T00:00:00'
class: jwst.pipeline.calwebb_spec2.Spec2Pipeline
parameters: {}
...
```

All of the keys under `meta` are required, most of which are self-explanatory. For more information, refer to the CRDS documentation (https://jwst-crds.stsci.edu/static/users_guide/).

The one keyword to explain further is `reftype`. This is what CRDS uses to determine which parameter file is being sought after. This has the format `pars-<step_name>` where `<step_name>` is the Python class name, in lowercase.

History

Parameter reference files also require at least one history entry. This can be found in the `history` block under `entries`:

```
history:
  entries:
  - !core/history_entry-1.0.0 {description: Base values, time: !!timestamp '2019-10-29
      21:20:50'}
```

It is highly suggested to use the ASDF API to add history entries:

```
>>> import asdf
>>> cfg = asdf.open('config.asdf')
    #
    # Modify `parameters` and `meta` as necessary.
    #
>>> cfg.add_history_entry('Parameters modified for some reason')
>>> cfg.write_to('config_modified.asdf')
```

JWST, Parameters and Parameter References

In general, the default parameters for any pipeline or step are valid for nearly all instruments and observing modes. This means that when a pipeline or step is run without any explicit parameter setting, that pipeline or step will usually do the desired operation. Hence, most of the time there is no need for a parameter reference to be available in CRDS, or provided by the user. Only for a small set of observing mode/step combinations, will there be need to create a parameter reference. Even then, nearly all cases will involve changing a subset of a pipeline or step parameters.

Keeping this sparse-population philosophy in mind, for most parameter references, only those parameters that are explicitly changed should be specified in the reference. If adhered to, when a pipeline/step default value for a particular parameter needs to change, the change will be immediately available. Otherwise, all references that mistakenly set said parameter will need to be updated. See *Completeness* for more information.

Furthermore, every pipeline/step have a common set of parameters, listed below. These parameters generally affect the infrastructure operation of pipelines/steps, and should not be included in a parameter reference.

- input_dir
- output_ext
- output_use_index

- output_use_model
- post_hooks
- pre_hooks
- save_results
- search_output_file

Configuration (CFG) Files

Note: The `cfg` format can still be used but is deprecated in favor of *ASDF Parameter Files*. Please convert any processes that use `cfg` files to the `ASDF` format. Note also that all `cfg` files that are currently being delivered in the package and retrieved using `collect_pipeline_cfgs` set no parameters; files are empty. All steps query CRDS parameter references for any data-dependent parameter settings, or use coded defaults.

The `cfg` format for configuration files uses the well-known ini-file format.

You can use the `collect_pipeline_cfgs` task to get copies of all the cfg files currently in use by the jwst pipeline software. The task takes a single argument, which is the name of the directory to which you want the cfg files copied. Use '.' to specify the current working directory, e.g.

```
$ collect_pipeline_cfgs .
```

Each step and pipeline has their own cfg file, which are used to specify relevant parameter values. For each step in a pipeline, the pipeline cfg file specifies either the step's arguments or the cfg file containing the step's arguments.

For a given step, the step's cfg file specifies parameters and their default values; it includes parameters that are typically not changed between runs. Parameters that are usually reset for each run are not included in the cfg file, but instead specified on the command line. An example of a cfg file for the jump detection step is:

```
name = "jump"
class = "jwst.jump.JumpStep"
rejection_threshold = 4.0
```

You can list all of the parameters for this step using:

```
$ strun jump.cfg -h
```

which gives the usage, the positional arguments, and the optional arguments.

Executing a pipeline or pipeline step via call()

The `call` method will create an instance and run a pipeline or pipeline step in a single call.

```
from jwst.pipeline import Detector1Pipeline
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits')

from jwst.linearity import LinearityStep
result = LinearityStep.call('jw00001001001_01101_00001_mirimage_uncal.fits')
```

To set custom parameter values when using the `call` method, set the parameters in the pipeline or parameter file and then supply the file using the `config_file` keyword:

```
# Calling a pipeline
result = Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits', config_
↪file='calwebb_detector1.asdf')

# Calling a step
result = LinearityStep.call('jw00017001001_01101_00001_nrca1_uncal.fits', config_file=
↪'linearity.asdf')
```

When running a pipeline, parameter values can also be supplied in the call to `call` itself by using a nested dictionary of step and parameter names:

```
result = Detector1Pipeline.call("jw00017001001_01101_00001_nrca1_uncal.fits", config_
↪file='calwebb_detector1.asdf', steps={"jump":{"rejection_threshold": 200}})
```

When running a single step with `call`, parameter values can be supplied more simply:

```
result = JumpStep.call("jw00017001001_01101_00001_nrca1_uncal.fits", rejection_
↪threshold=200)
```

Running steps and pipelines with `call` also allows for the specification of a logging configuration file using the keyword `logcfg`:

```
result = Detector1Pipeline.call("jw00017001001_01101_00001_nrca1_uncal.fits",
                                config_file="calwebb_detector1.asdf",
                                logcfg="my-logging-config.cfg")
```

Note that naming the logging configuration file "stpipe-log.cfg" will configure logging without assignment of the `logcfg` keyword, as `stpipe` searches for this filename in the local directory during execution. If the configuration should be used only when specified, ensure your file is named something other than "stpipe-log.cfg"!

Where are the results?

A fundamental difference between running steps and pipelines in Python as opposed to from the command line using `strun` is whether files are created or not. When using `strun`, results are automatically saved to files because that is the only way to access the results.

However, when running within a Python interpreter or script, the presumption is that results will be used within the script. As such, results are not automatically saved to files. It is left to the user to decide when to save.

If one wishes for results to be saved by a particular `call`, use the parameter `save_results=True`:

```
result = JumpStep.call("jw00017001001_01101_00001_nrca1_uncal.fits",
                       rejection_threshold=200, save_results=True)
```

If one wishes to specify a different file name, rather than a system-generated one, set *output_file* and/or *output_dir*.

Executing a pipeline or pipeline step directly, or via run()

When calling a pipeline or step instance directly, or using the `run` method, you can specify individual parameter values manually. In this case, parameter files are not used. If you use `run` after instantiating with a parameter file (as is done when using the *call* method), the parameter file will be ignored.

```
# Instantiate the class. Do not provide a parameter file.
pipe = Detector1Pipeline()

# Manually set any desired non-default parameter values
pipe.refpix.skip = True
pipe.jump.rejection_threshold = 5
pipe.ramp_fit.override_gain = 'my_gain_file.fits'
pipe.save_result = True
pipe.output_dir = '/my/data/pipeline_outputs'

# Run the pipeline
result = pipe('jw00017001001_01101_00001_nrca1_uncal.fits')

# Or, execute the pipeline using the run method
result = pipe.run('jw00017001001_01101_00001_nrca1_uncal.fits')
```

To run a single step:

```
from jwst.jump import JumpStep

# Instantiate the step
step = JumpStep()

# Set parameter values
step.rejection_threshold = 5
step.save_results = True
step.output_dir = '/my/data/jump_data'

# Execute by calling the instance directly
result = step('jw00017001001_01101_00001_nrca1_linearity.fits')

# Or, execute using the run method
result = step.run('jw00017001001_01101_00001_nrca1_linearity.fits')
```

Parameter Files

Parameter files can be used to specify parameter values when running a pipeline or individual steps. For JWST, parameter files are retrieved from CRDS, just as with other reference files. If there is no match between a step, the input data, and CRDS, the coded defaults are used. These values can be overridden either by the command line options and/or a local parameter file. See *Parameter Precedence* for a full description of how a parameter gets its final value.

Note: Retrieval of `Step` parameters from CRDS can be completely disabled by using the `--disable-crds-steppars` command-line switch, or setting the environment variable `STPIPE_DISABLE_CRDS_STEPPARS` to `true`.

A parameter file should be used when there are parameters a user wishes to change from the default/CRDS version for a custom run of the step. To create a parameter file add `--save-parameters <filename.asdf>` to the command:

```
$ strun <step.class> <required-input-files> --save-parameters <filename.asdf>
```

For example, to save the parameters used for a run of the `calwebb_image2` pipeline, use:

```
$ strun calwebb_image2 jw82500001003_02101_00001_NRCALONG_rate.fits --save-parameters my_
↪image2.asdf
```

Once saved, the file can be edited, removing parameters that should be left at their default/CRDS values, and setting the remaining parameters to the desired values. Once modified, the new parameter file can be used:

```
$ strun my_image2.asdf jw82500001003_02101_00001_NRCALONG_rate.fits
```

Note that the parameter values will reflect whatever was set on the command-line, through a specified local parameter file, and what was retrieved from CRDS. In short, the values will be those actually used in the running of the step.

For more information about and editing of parameter files, see *ASDF Parameter Files*. Note that the older *Configuration (CFG) Files* format is still an option, understanding that this format will be deprecated.

More information on parameter files can be found in the `stpipe` User's Guide at *For Users*.

CFG Usage Deprecation Notice

As of March 18, 2021, a significant change to how JWST pipelines operate was completed and pushed to the JWST master branch on github. Theoretically the change should be transparent. However, we are all familiar with the difference between theory and practice and hence we want to alert all users.

Originally, how the pipelines operated was determined by a set of configuration (CFG) files that were delivered as part of the JWST package. These configuration files were retrieved using the `collect_pipeline_cfgs` command. The configuration files were used to run each of the different pipelines using the `strun` command. For example:

```
$ collect_pipeline_cfgs ./
$ strun calwebb_spec2.cfg an_exposure_file.fits
```

The issue with the above process is that any changes, as determined by INS and the Calibration Working Group, to the default operation of the pipeline requires a code release. A better solution would be if the pipeline configurations could come from reference files retrieved from CRDS.

As of the version of master introduced on March 18th, 2021, in conjunction with CRDS context jwst_0712, the default pipeline configurations no longer depend on the package-delivered configuration files. Instead, all default configuration relies on settings in the pipeline code itself, using CRDS-retrieved parameter reference files to modify any parameters that are data-dependent. There is no longer any need to run `collect_pipeline_cfgs` and specify a configuration file for the `strun` command. One only needs to specify a simplified pipeline name. In most cases, this simple name, or alias, is the same as the name of the old configuration file, but without the suffix `.cfg`.

Taking the example above, to get the same operation, the single command would become:

```
$ strun calwebb_spec2 an_exposure_file.fits
```

The JWST documentation has been updated to account for this change in usage. To get familiarized, it is best to start with the *Introduction*

A list of the available pipeline aliases can be found in the *Pipeline Stages* section.

An added benefit to removing the dependency on package-delivered configuration files is that users, under normal circumstances, no longer need to be concerned with configuration files and whether they are up-to-date. One only needs to install the JWST package and start using the pipelines out-of-the-box.

Does this mean that everyone has to immediately change their behavior and code if using the default configuration files? Short answer is "No". If one wishes to continue using the package-delivered configuration files from `collect_pipeline_cfgs`, one may do so. However, these configuration files no longer contain any parameter settings; only the class name of the pipeline to be run. This allows the code-plus-CRDS-retrieved parameter reference files to determine operation.

Since the configuration settings have simply been moved to CRDS, the results one obtains should not change. If a change in behavior is noted, please report the issue to the Help Desk, file a Github issue on the JWST Github repository, or file a Jira issue against the JP project.

In the meantime, please consider deprecating the use of `collect_pipeline_cfgs` and the .cfg files in favor of simply specifying pipeline aliases, as the documentation now describes.

For users that use their own, custom configuration files, there is no change to functionality. However, there are changes to both how these files are documented and their format.

Concerning documentation, there is a change of terminology. No longer are these files referred to as "configuration files", but are called "parameter files" or "parameter reference files" when retrieved from CRDS.

In order to simplify integration with CRDS, the format of parameter files have changed from the "cfg", init-like format, to the ASDF format. All parameter files in CRDS are in this format. Similarly, the tools provided by the JWST package to create parameter files will create them in ASDF. "cfg"-formatted files are still supported, but it is strongly suggested that users change to using the ASDF form. For more information, please to refer to *ASDF Parameter Files*

As always, if anyone finds any discrepancies or other issues with the documentation, or actual operation of the pipelines, please contact the Help Desk, or file issues directly against the Github repository or the JIRA "JP" project.

For Developers

Steps

Writing a step

Writing a new step involves writing a class that has a `process` method to perform work and a `spec` member to define its configuration parameters. (Optionally, the `spec` member may be defined in a separate `spec` file).

Inputs and outputs

A `Step` provides a full framework for handling I/O. Below is a short description. A more detailed discussion can be found in *Step I/O Design*.

Steps get their inputs from two sources:

- Configuration parameters come from the parameter file or commandline and are set as member variables on the Step object by the stpipe framework.
- Arguments are passed to the Step's `process` function as regular function arguments.

Parameters should be used to specify things that must be determined outside of the code by a user using the class. Arguments should be used to pass data that needs to go from one step to another as part of a larger pipeline. Another way to think about this is: if the user would want to examine or change the value, use a parameter.

The parameters are defined by the *Step.spec* member.

Input Files, Associations, and Directories

All input files must be in the same directory. This directory is whichever directory the first input file is found in. This is particularly important for associations. It is assumed that all files referenced by an association are in the same directory as the association file itself.

Output Files and Directories

The step will generally return its output as a data model. Every step has implicitly created parameters `output_dir` and `output_file` which the user can use to specify the directory and file to save this model to. Since the `stpipe` architecture generally creates output file names, in general, it is expected that `output_file` be rarely specified, and that different sets of outputs be separated using `output_dir`.

Output Suffix

There are three ways a step's results can be written to a file:

1. Implicitly when a step is run from the command line or with `Step.from_cmdline`
2. Explicitly by specifying the parameter `save_results`
3. Explicitly by specifying a file name with the parameter `output_file`

In all cases, the file, or files, is/are created with an added suffix at the end of the base file name. By default this suffix is the class name of the step that produced the results. Use the `suffix` parameter to explicitly change the suffix.

For pipelines, this can be done either in a parameter file, or within the code itself. See *calwebb_dark* for an example of specifying in the parameter file.

For an example where the suffix can only be determined at runtime, see *calwebb_detector1*. For an example of a pipeline that returns many results, see *calwebb_spec2*.

The Python class

At a minimum, the Python Step class should inherit from `stpipe.Step`, implement a `process` method to do the actual work of the step and have a `spec` member to describe its parameters.

1. Objects from other Steps in a pipeline are passed as arguments to the `process` method.
2. The parameters described in *Configuring a Step* are available as member variables on `self`.
3. To support the caching suspend/resume feature of pipelines, images must be passed between steps as model objects. To ensure you're always getting a model object, call the model constructor on the parameter passed in. It is good idea to use a `with` statement here to ensure that if the input is a file path that the file will be appropriately closed.
4. Use `get_reference_file_model` method to load any CRDS reference files used by the Step. This will make a cached network request to CRDS. If the user of the step has specified an override for the reference file in either the parameter file or at the command line, the override file will be used instead. (See *Interfacing with CRDS*).
5. Objects to pass to other Steps in the pipeline are simply returned from the function. To return multiple objects, return a tuple.
6. The parameters for the step are described in the `spec` member in the `configspec` format.
7. Declare any CRDS reference files used by the Step. (See *Interfacing with CRDS*).

```
from jwst.stpipe import Step

from stdatamodels.jwst.datamodels import ImageModel
from my_awesome_astronomy_library import combine

class ExampleStep(Step):
    """
    Every step should include a docstring.  This docstring will be
    displayed by the `strun --help`.
    """

    # 1.
    def process(self, image1, image2):
        self.log.info("Inside ExampleStep")

        # 2.
        threshold = self.threshold

        # 3.
        with ImageModel(image1) as image1, ImageModel(image2) as image2:
            # 4.
            with self.get_reference_file_model(image1, "flat_field") as flat:
                new_image = combine(image1, image2, flat, threshold)

        # 5.
        return new_image

   # 6.
   spec = """
   # This is the configspec file for ExampleStep

   threshold = float(default=1.0)  # maximum flux
   """

   # 7.
   reference_file_types = ['flat_field']
```

The Python Step subclass may be installed anywhere that your Python installation can find it. It does not need to be installed in the `stpipe` package.

The spec member

The `spec` member variable is a string containing information about the parameters. It is in the `configspec` format defined in the `ConfigObj` library that stpipe uses.

The `configspec` format defines the types of the parameters, as well as allowing an optional tree structure.

The types of parameters are declared like this:

```
n_iterations = integer(1, 100)  # The number of iterations to run
factor = float()                # A multiplication factor
author = string()               # The author of the file
```

Note that each parameter may have a comment. This comment is extracted and displayed in help messages and docstrings etc.

Parameters can be grouped into categories using ini-file-like syntax:

```
[red]
offset = float()
scale = float()

[green]
offset = float()
scale = float()

[blue]
offset = float()
scale = float()
```

Default values may be specified on any parameter using the `default` keyword argument:

```
name = string(default="John Doe")
```

While the most commonly useful parts of the configspec format are discussed here, greater detail can be found in the configspec documentation (https://configobj.readthedocs.io/en/latest/).

Configspec types

The following is a list of the commonly useful configspec types.

`integer`: matches integer values. Takes optional `min` (https://docs.python.org/3/library/functions.html#min) and `max` (https://docs.python.org/3/library/functions.html#max) arguments:

```
integer()
integer(3, 9)  # any value from 3 to 9
integer(min=0) # any positive value
integer(max=9)
```

`float` (https://docs.python.org/3/library/functions.html#float): matches float values Has the same parameters as the integer check.

`boolean`: matches boolean values: True or False.

`string` (https://docs.python.org/3/library/string.html#module-string): matches any string. Takes optional keyword args `min` (https://docs.python.org/3/library/functions.html#min) and `max` (https://docs.python.org/3/library/functions.html#max) to specify min and max length of string.

`list` (https://docs.python.org/3/library/stdtypes.html#list): matches any list. Takes optional keyword args `min` (https://docs.python.org/3/library/functions.html#min), and `max` (https://docs.python.org/3/library/functions.html#max) to specify min and max sizes of the list. The list checks always return a list.

`force_list`: matches any list, but if a single value is passed in will coerce it into a list containing that value.

`int_list`: Matches a list of integers. Takes the same arguments as list.

`float_list`: Matches a list of floats. Takes the same arguments as list.

`bool_list`: Matches a list of boolean values. Takes the same arguments as list.

`string_list`: Matches a list of strings. Takes the same arguments as list.

`option`: matches any from a list of options. You specify this test with:

```
option('option 1', 'option 2', 'option 3')
```

Normally, steps will receive input files as parameters and return output files from their process methods. However, in cases where paths to files should be specified in the parameter file, there are some extra parameter types that stpipe provides that aren't part of the core configobj library.

`input_file`: Specifies an input file. Relative paths are resolved against the location of the parameter file. The file must also exist.

`output_file`: Specifies an output file. Identical to `input_file`, except the file doesn't have to already exist.

Interfacing with CRDS

If a Step uses CRDS to retrieve reference files, there are two things to do:

1. Within the `process` method, call `self.get_reference_file` or `self.get_reference_file_model` to get a reference file from CRDS. These methods take as input a) a model for the input file, whose metadata is used to do a CRDS bestref lookup, and b) a reference file type, which is just a string to identify the kind of reference file.
2. Declare the reference file types used by the Step in the `reference_file_types` member. This information is used by the stpipe framework for two purposes: a) to pre-cache the reference files needed by a Pipeline before any of the pipeline processing actually runs, and b) to add override parameters to the Step's configspec.

For each reference file type that the Step declares, an `override_*` parameter is added to the Step's configspec. For example, if a step declares the following:

```
reference_file_types = ['flat_field']
```

then the user can override the flat field reference file using the parameter file:

```
override_flat_field = /path/to/my_reference_file.fits
```

or at the command line:

```
--override_flat_field=/path/to/my_reference_file.fits
```

Making a simple commandline script for a step

Any step can be run from the commandline using *Running a Step from the commandline*. However, to make a step even easier to run from the commandline, a custom script can be created. stpipe provides a function `stpipe.cmdline.step_script` to make those scripts easier to write.

For example, to make a script for the step `mypackage.ExampleStep`:

```
#!/usr/bin/python
# ExampleStep

# Import the custom step
from mypackage import ExampleStep
```

(continues on next page)

(continued from previous page)

```
# Import stpipe.cmdline
from jwst.stpipe import cmdline

if __name__ == '__main__':
    # Pass the step class to cmdline.step_script
    cmdline.step_script(ExampleStep)
```

Running this script is similar to invoking the step with *Running a Step from the commandline*, with one difference. Since the Step class is known (it is hard-coded in the script), it does not need to be specified on the commandline. To specify a config file on the commandline, use the `--config-file` option.

For example:

```
> ExampleStep

> ExampleStep --config-file=example_step.asdf

> ExampleStep --parameter1=42.0 input_file.fits
```

Pipelines

Writing a Pipeline

The basics of writing a Pipeline are just like *Writing a step*, but instead of inheriting from the `Step` class, one inherits from the `Pipeline` class.

In addition, a Pipeline subclass defines what its Steps are so that the framework can configure parameters for the individual Steps. This is done with the `step_defs` member, which is a dictionary mapping step names to step classes. This dictionary defines what the Steps are, but says nothing about their order or how data flows from one Step to the next. That is defined in Python code in the Pipeline's `process` method. By the time the Pipeline's `process` method is called, the Steps in `step_defs` will be instantiated as member variables.

For example, here is a pipeline with two steps: one that processes each chip of a multi-chip FITS file, and another to combine the chips into a single image:

```
from jwst.stpipe import Pipeline

from stdatamodels.jwst.datamodels import ImageModel

# Some locally-defined steps
from . import FlatField, Combine

class ExamplePipeline(Pipeline):
    """
    This example pipeline demonstrates how to combine steps
    using Python code, in some way that it not necessarily
    a linear progression.
    """

    step_defs = {
        'flat_field': FlatField,
```

(continues on next page)

(continued from previous page)

```
        'combine': Combine,
        }

    def process(self, input):
        with ImageModel(input) as science:

            flattened = self.flat_field(science, self.multiplier)

            combined = self.combine(flattened)

        return combined

    spec = """
    multiplier = float()     # A multiplier constant
    """
```

When writing the spec member for a Pipeline, only the parameters that apply to the Pipeline as a whole need to be included. The parameters for each Step are automatically loaded in by the framework.

In the case of the above example, we define two new pipeline parameters for the flat field file and the output filename.

The parameters for the individual substeps that make up the Pipeline will be implicitly added by the framework.

Logging

The logging in stpipe is built on the Python standard library's `logging` (https://docs.python.org/3/library/logging.html#module-logging) module. For detailed information about logging, refer to the documentation there. This document basically outlines some simple conventions to follow so that the configuration mechanism described in *Logging* works.

Logging from a Step or Pipeline

Each Step instance (and thus also each Pipeline instance) has a `log` member, which is a Python `logging.Logger` (https://docs.python.org/3/library/logging.html#logging.Logger) instance. All messages from the Step should use this object to log messages. For example, from a `process` method:

```
self.log.info("This Step wants to say something")
```

Logging from library code

Often, you may want to log something from code that is oblivious to the concept of stpipe Steps. In that case, stpipe has special code that allows library code to use any logger and have those messages appear as if they were coming from the step that used the library. All the library code has to do is use a Python `logging.Logger` (https://docs.python.org/3/library/logging.html#logging.Logger) as normal:

```
import logging

# ...
log = logging.getLogger()
```

(continues on next page)

(continued from previous page)

```
# If the log on its own won't emit, neither will it in the
# context of an stpipe step, so make sure the level is set to
# allow everything through
log.setLevel(logging.DEBUG)

def my_library_call():
    # ...
    log.info("I want to make note of something")
    # ...
```

Step I/O Design

API Summary

`Step` command-line options

- `--output_dir`: *Directory* where all output will go.
- `--output_file`: *File name* upon which output files will be based.

`Step` configuration options

- `output_dir`: *Directory* where all output will go.
- `output_file`: *File name* upon which output files will be based.
- `suffix`: *Suffix* defining the output of this step.
- `save_results`: True to create output files. *[more]*
- `search_output_file`: True to retrieve the `output_file` from a parent `Step` or `Pipeline`. *[more]*
- `output_use_model`: True to always base output file names on the `DataModel.meta.filename` of the `DataModel` being saved.
- `input_dir`: Generally defined by the location of the primary input file unless otherwise specified. All input files must be in this directory.

Classes, Methods, Functions

- `Step.open_model`: Open a `DataModel`
- `Step.load_as_level2_asn()`: Open a list or file as Level2 association.
- `Step.load_as_level3_asn()`: Open a list or file as Level3 association.
- `Step.make_input_path`: Create a file name to be used as input
- `Step.save_model`: Save a `DataModel` immediately
- `Step.make_output_path`: Create a file name to be used as output

Design

The `Step` architecture is designed such that a `Step`'s intended sole responsibility is to perform the calculation required. Any input/output operations are handled by the surrounding `Step` architecture. This is to help facilitate the use of `Step`'s from both a command-line environment, and from an interactive Python environment, such as Jupyter notebooks or `ipython`.

For command-line usage, all inputs and outputs are designed to come from and save to files.

For interactive Python use, inputs and outputs are expected to be Python objects, negating the need to save and reload data after every `Step` call. This allows users to write Python scripts without having to worry about doing I/O at every step, unless, of course, if the user wants to do so.

The high-level overview of the input/output design is given in *Writing a step*. The following discusses the I/O API and best practices.

To facilitate this design, a basic `Step` is suggested to have the following structure:

```
class MyStep(jwst.stpipe.step.Step):

    spec = ''  # Desired configuration parameters

    def process(self, input):

        with jwst.datamodels.open(input) as input_model:

            # Do awesome processing with final result
            # in `result`
            result = final_calculation(input_model)

        return (result)
```

When run from the command line:

```
strun MyStep input_data.fits
```

the result will be saved in a file called:

```
input_data_mystep.fits
```

Similarly, the same code can be used in a Python script or interactive environment as follows:

```
>>> import jwst
>>> input = jwst.datamodels.open('input_data.fits')
>>> result = MyStep.call(input)
    # `result` contains the resulting data
    # which can then be used by further `Steps`'s or
    # other functions.
    #
    # when done, the data can be saved with the `DataModel.save`
    # method
>>> result.save('my_final_results.fits')
```

Input and JWST Conventions

A `Step` gets its input from two sources:

- Configuration parameters
- Arguments to the `Step.process` method

The definition and use of parameters is documented in *Writing a step*.

When using the `Step.process` arguments, the code must at least expect strings. When invoked from the command line using `strun`, how many arguments to expect are the same number of arguments defined by `Step.process`. Similarly, the arguments themselves are passed to `Step.process` as strings.

However, to facilitate code development and interactive usage, code is expected to accept other object types as well.

A `Step`'s primary argument is expected to be either a string containing the file path to a data file, or a JWST `JwstDataModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.JwstDataModel.html#jwst.datamodels.JwstDataMod object. The method `open_model()` handles either type of input, returning a `DataModel` from the specified file or a shallow copy of the `DataModel` that was originally passed to it. A typical pattern for handling input arguments is:

```
class MyStep(jwst.stpipe.step.Step):

    def process(self, input_argument):

        input_model = self.open_model(input_argument)

        ...
```

`input_argument` can either be a string containing a path to a data file, such as `FITS` file, or a `DataModel` directly.

`open_model()` handles `Step`-specific issues, such ensuring consistency of input directory handling.

If some other file type is to be opened, the lower level method `make_input_path()` can be used to specify the input directory location.

Input and Associations

Many of the JWST calibration steps and pipelines expect an *Association* file as input. When opened with `open_model()`, a `ModelContainer` is returned. `ModelContainer` is, among other features, a list-like object where each element is the `DataModel` of each member of the association. The `meta.asn_table` is populated with the association data structure, allowing direct access to the association itself. The association file, as well as the files listed in the association file, must be in the input directory.

To read in a list of files, or an association file, as an association, use the `load_as_level2_asn` or `load_as_level3_asn` methods.

Input Source

All input files, except for references files provided by CRDS, are expected to be co-resident in the same directory. That directory is determined by the directory in which the primary input file resides. For programmatic use, this directory is available in the `Step.input_dir` attribute.

Output

When Files are Created

Whether a `Step` produces an output file or not is ultimately determined by the built-in parameter option `save_results`. If `True` (https://docs.python.org/3/library/constants.html#True), output files will be created. `save_results` is set under a number of conditions:

- Explicitly through a parameter file or as a command-line option.
- Implicitly when a step is called by `strun`.

Output File Naming

File names are constructed based on three components: basename, suffix, and extension:

```
basename_suffix.extension
```

The extension will often be the same as the primary input file. This will not be the case if the data format of the output needs to be something different, such as a text table with `ecsv` extension.

Similarly, the basename will usually be derived from the primary input file. However, there are some *caveats* discussed below.

Ultimately, the suffix is what `Step`'s use to identify their output. The most common suffixes are listed in the *Pipeline/Step Suffix Definitions*.

A `Step`'s suffix is defined in a couple of different ways:

- By the `Step.name` attribute. This is the default.
- By the `suffix` parameter.
- Explicitly in the code. Often this is done in `Pipelines` where a single pipeline creates numerous different output files.

Basename Determination

Most often, the output file basename is determined through any of the following, given from higher precedence to lower:

- The `--output_file` command-line option.
- The `output_file` parameter option.
- Primary input file name.
- If the output is a `DataModel`, from the `DataModel.meta.filename`.

In all cases, if the originating file name has a known suffix on it, that suffix is removed and replaced by the `Step`'s own suffix.

In very rare cases, when there is no other source for the basename, a basename of `step_<step_name>` is used. This can happen when a `Step` is being programmatically used and only the `save_results` parameter option is given.

Sub-Steps and Output

Normally, the value of a parameter option is completely local to the `Step`: A `Step`, called from another `Step` or `Pipeline`, can only access its own parameters. Hence, options such as `save_results` do not affect a called `Step`.

The exceptions to this are the parameters `output_file` and `output_dir`. If either of these parameters are queried by a `Step`, but are not defined for that `Step`, values will be retrieved up through the parent. The reason is to provide consistency in output from `Step` and `Pipelines`. All file names will have the same basename and will all appear in the same directory.

As expected, if either parameter is specified for the `Step` in question, the local value will override the parent value.

Also, for `output_file`, there is another option, `search_output_file`, that can also control this behavior. If set to `False` (https://docs.python.org/3/library/constants.html#False), a `Step` will never query its parent for its value.

Basenames, Associations, and Stage 3 Pipelines

Stage 3 pipelines, such as *calwebb_image3* or *calwebb_spec3*, take associations as their primary input. In general, the association defines what the output basename should be. A typical pattern used to handle associations is:

```
class MyStep(jwst.stpipe.step.Step):

    spec = ''  # Desired configuration parameters

    def process(self, input):

        with jwst.datamodels.open(input) as input_model:

            # If not already specified, retrieve the output
            # file name from the association.
            if self.save_results and self.output_file is None:
                try:
                   self.output_file = input_model.meta.asn_table.products[0].name

                except AttributeError:
                    pass

            # Do awesome processing with final result
            # in `result`
            result = final_calculation(input_model)

        return (result)
```

Some pipelines, such as *calwebb_spec3*, call steps which are supposed to save their results, but whose basenames should not be based on the association product name. An example is the *OutlierDetectionStep* step. For such steps, one can prevent using the `Pipeline.output_file` specification by setting the parameter `search_output_file=False`. When such steps then save their output, they will go through the standard basename search. If nothing else is specified,

the basename will be based on `DataModel.meta.filename` that step's result, creating appropriate names for that step.

Output API: When More Control Is Needed

In summary, the standard output API, as described so far, is basically "set a few parameters, and let the `Step` framework handle the rest". However, there are always the exceptions that require finer control, such as saving intermediate files or multiple files of different formats. This section discusses the method API and conventions to use in these situations.

Save That Model: Step.save_model

If a `Step` needs to save a `DataModel` before the step completes, use of `Step.save_model` is the recommended over directly calling `DataModel.save`. `Step.save_model` uses the `Step` framework and hence will honor the following:

- If `Step.save_results` is `False` (https://docs.python.org/3/library/constants.html#False), nothing will happen.
- Will ensure that `Step.output_dir` is used.
- Will use `Step.suffix` if not otherwise specified.
- Will determine the output basename through the `Step` framework, if not otherwise specified.

The basic usage, in which nothing is overridden, is:

```
class MyStep(Step):

    def process(self, input):
        ...
        result = some_DataModel
        self.save_model(result)
```

The most common use case, however, is for saving some intermediate results that would have a different suffix:

```
self.save_model(intermediate_result_datamodel, suffix='intermediate')
```

See `jwst.stpipe.step.Step.save_model()` for further information.

Make That Filename: Step.make_output_path

For the situations when a filename is needed to be constructed before saving, either to know what the filename will be or for data that is not a `DataModel`, use `Step.make_output_path`. By default, calling `make_output_path` without any arguments will return what the default output file name will be:

```
output_path = self.make_output_path()
```

This method encapsulates the following `Step` framework functions:

- Will ensure that `Step.output_dir` is used.
- Will use `Step.suffix` if not otherwise specified.
- Will determine the output basename through the `Step` framework, if not otherwise specified.

A typical use case is when a `Step` needs to save data that is not a `DataModel`. The current `Step` architecture does not know how to handle these, so saving needs to be done explicitly. The pattern of usage would be:

```
# A table need be saved and needs a different
# suffix than what the Step defines.
table = some_astropy_table_data
if self.save_results:
    table_path = self.make_output_path(suffix='cat', ext='ecsv')
    table.save(table_path, format='ascii.ecsv', overwrite=True)
```

jwst.stpipe Package

Classes

Step	alias of JwstStep
Pipeline	alias of JwstPipeline

Step

jwst.stpipe.**Step**

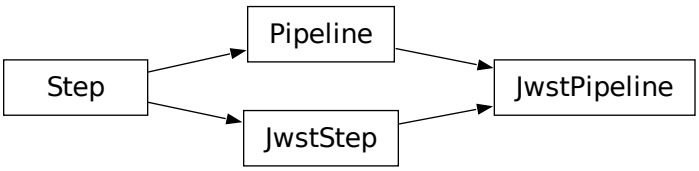
alias of JwstStep

Pipeline

jwst.stpipe.**Pipeline**

alias of JwstPipeline

Class Inheritance Diagram



15.1.60 Stray Light Correction

Description

Class
jwst.straylight.StraylightStep

Alias
straylight

Assumption

The `straylight` correction is only valid for MIRI MRS data.

Overview

This routine removes contamination of MIRI MRS spectral data by the MIRI cross artifact feature produced by internal reflections within the detector arrays. As discussed in depth for the MIRI Imager by A. Gáspár et al. 2021 (PASP, 133, 4504), the cross artifact manifests as a signal extending up to hundreds of pixels along the detector column and row directions from bright sources. This signal has both smooth and structured components whose profiles vary as a function of wavelength. Although the peak intensity of the cross artifact is at most 1% of the source intensity in Channel 1 (decreasing toward longer wavelengths), the total integrated light in this feature can be of order 20% of the total light from a given source.

In the MIRI MRS, such a signal extending along detector rows is more disruptive than for the MIRI imager. Since the individual IFU slices are interleaved on the detector and staggered in wavelength from each other, the cross artifact signal thus contaminates non-local regions in reconstructed data cubes (both non-local on the sky and offset in wavelength space from bright emission lines). The purpose of this routine is thus to model the cross artifact feature in a given science exposure and subtract it at the detector level prior to reformatting the data into three-dimensional cubes.

At the same time, this step also ensures that the median count rate (in DN/s) in regions of the detector that see no direct light from the sky is zero for consistency with the applied flux calibration vectors.

Algorithm

The basic idea of the cross artifact correction is to convolve a given science detector image with a kernel function that has been pre-calibrated based on observations of isolated sources and subtract the corresponding convolved image. As such, there are no free parameters in this step when applied to science data.

In Channel 1, the kernel function is based on engineering observations of isolated bright stars and consists of a broad low-amplitude Lorentzian function plus two pairs of double Gaussians. The low-amplitude Lorentzian describes the broad wings of the kernel, and typically has a FWHM of 100 pixels or more:

$$f_{Lor} = \frac{A_{Lor}\gamma^2}{\gamma^2 + (x - x_0)^2}$$

where $\gamma = FWHM/2$ and x_0 is the column coordinate of a given pixel.

The two double Gaussian functions describe the structured component of the profile, in which two peaks are seen on each side of a bright spectral trace on the detector. The relative offsets of these Gaussians (dx) are observed to be fixed with respect to each other, with the separation of the secondary Gaussian from the bright trace being double the separation of the first Gaussian and both increasing as a function of wavelength. The widths of the Gaussians (σ) are also tied, with the secondary Gaussian having double the width of the first. The inner Gaussians are thus described by:

$$f_{G1} = A_{G1} exp^{\frac{-(x-x_0-dx)^2}{2\sigma^2}}$$

$$f_{G3} = A_{G1} exp^{\frac{-(x-x_0+dx)^2}{2\sigma^2}}$$

while the outer Gaussians are described by:

$$f_{G2} = A_{G2} exp^{\frac{-(x-x_0-2dx)^2}{8\sigma^2}}$$

$$f_{G4} = A_{G2} exp^{\frac{-(x-x_0+2dx)^2}{8\sigma^2}}$$

The best-fit parameters of these models derived from engineering data are recorded in the *MRSXARTCORR* reference file and applied in a pixelwise manner to the detector data.

The kernel functions for Channels 2 and 3 rely upon engineering observations of bright extended sources, as the magnitude of the correction is typically too small to be visible from point sources. These channels use only a Lorentzian kernel with the Gaussian amplitudes set to zero as such structured components are less obvious at these longer wavelengths. In Channel 4 no correction appears to be necessary, and the amplitudes of all model components are set equal to zero.

Step Arguments

The `straylight` step has no step-specific arguments.

Reference Files

The `straylight` step uses the *MRSXARTCORR* reference file, which stores vectors describing the appropriate cross-artifact convolution kernel for each MRS band. In Channel 1 these vectors include power in a broad Lorentzian core plus a pair of double-Gaussian profiles. In Channels 2 and 3 these vectors include only power in the broad Lorentzian, while in Channel 4 there is no correction.

MRSXARTCORR reference file

REFTYPE
MRSXARTCORR

Data models
`MirMrsXArtCorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MirMrsXArtCorrModel.html#jv

The MRSXARTCORR reference file contains parameter values used to model and subtract the cross-artifact in the straylight step.

Reference Selection Keywords for MRSXARTCORR

CRDS selects appropriate MRSXARTCORR references based on the following keywords. MRSXARTCORR is not applicable for instruments not in the table.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for MRSXARTCORR

In addition to the standard reference file keywords listed above, the following keywords are *required* in MRSXARTCORR reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for MRSXARTCORR*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

MIRI Reference File Format

The MIRI MRSXARTCORR reference files are FITS format, with 12 BINTABLE extensions. The FITS primary data array is assumed to be empty. The format and content of the MIRI MRSXARTCORR reference file

EXTNAME	XTENSION	Dimensions
1A	BINTABLE	TFIELDS = 7
1B	BINTABLE	TFIELDS = 7
1C	BINTABLE	TFIELDS = 7
2A	BINTABLE	TFIELDS = 7
2B	BINTABLE	TFIELDS = 7
2C	BINTABLE	TFIELDS = 7
3A	BINTABLE	TFIELDS = 7
3B	BINTABLE	TFIELDS = 7
3C	BINTABLE	TFIELDS = 7
4A	BINTABLE	TFIELDS = 7
4B	BINTABLE	TFIELDS = 7
4C	BINTABLE	TFIELDS = 7

The formats of the individual table extensions are listed below.

Table	Column	Data type	Units
CH1A	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH1B	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH1C	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH2A	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH2B	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH2C	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH3A	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A

continues on next page

Table 9 – continued from previous page

Table	Column	Data type	Units
CH3B	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH3C	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH4A	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH4B	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A
CH4C	YROW	shortint	pixels
	LOR_FWHM	float	pixels
	LOR_SCALE	float	N/A
	GAU_FWHM	float	pixels
	GAU_XOFF	float	pixels
	GAU_SCALE1	float	N/A
	GAU_SCALE2	float	N/A

These reference files contain tables for each wavelength band giving the appropriate kernel properties to use to model the cross-artifact for each band. These include Lorentzian plus Gaussian models.

The MIRI reference table contains parameters for each band in the corresponding EXTNAME extension.

jwst.straylight Package

Classes

StraylightStep([name, parent, config_file, ...])	StraylightStep: Performs straylight correction image using a Mask file.

StraylightStep

class jwst.straylight.**StraylightStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

StraylightStep: Performs straylight correction image using a Mask file.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'straylight'

reference_file_types = ['mrsxartcorr']

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.61 Superbias Subtraction

Description

Class
jwst.superbias.SuperBiasStep

Alias
superbias

The superbias subtraction step removes the fixed detector bias from a science data set by subtracting a superbias reference image.

Algorithm

The 2-D superbias reference image is subtracted from every group in every integration of the input science ramp data. Any NaN's that might be present in the superbias image are set to a value of zero before being subtracted from the science data, such that those pixels effectively receive no correction.

The DQ array from the superbias reference file is combined with the science exposure "PIXELDQ" array using a bitwise OR operation.

The ERR and GROUPDQ arrays in the science ramp data are unchanged.

NIRCam Frame 0

If the NIRCam frame zero data cube is present in the input data, the image for each integration has the superbias reference image subtracted from it, in the same way as the regular science data.

Subarrays

If the subarray mode of the superbias reference file matches that of the science exposure, the reference data are directly subtracted. If the superbias reference file contains full-frame data, while the science exposure is a subarray mode, the corresponding subarray is extracted from the superbias reference data before being subtracted.

NIRSpec IRS2

No special handling is necessary for science data taken in the IRS2 readout mode, because matching IRS2 superbias reference files are supplied in CRDS.

Step Arguments

The superbias subtraction step has no step-specific arguments.

Reference Files

The `superbias` subtraction step uses a SUPERBIAS reference file.

SUPERBIAS Reference File

REFTYPE
SUPERBIAS

Data model
`SuperBiasModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.SuperBiasModel.html#jwst.datamodels

The SUPERBIAS reference file contains a 2-D image of the detector bias ("zeroth" read) structure.

Reference Selection Keywords for SUPERBIAS

CRDS selects appropriate SUPERBIAS references based on the following keywords. SUPERBIAS is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
FGS	INSTRUME, DETECTOR, READPATT, SUBARRAY, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, DETECTOR, READPATT, SUBARRAY, DATE-OBS, TIME-OBS
NIRISS	INSTRUME, DETECTOR, READPATT, SUBARRAY, DATE-OBS, TIME-OBS
NIRSpec	INSTRUME, DETECTOR, READPATT, SUBARRAY, SUBSTRT1, SUBSTRT2, SUBSIZE1, SUBSIZE2, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for SUPERBIAS

In addition to the standard reference file keywords listed above, the following keywords are *required* in SUPERBIAS reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for SUPERBIAS*):

Keyword	Data Model Name	Instruments
DETECTOR	model.meta.instrument.detector	FGS, NIRCam, NIRISS, NIRSpec
READPATT	model.meta.exposure.readpatt	FGS, NIRCam, NIRISS, NIRSpec
SUBARRAY	model.meta.subarray.name	FGS, NIRCam, NIRISS, NIRSpec
SUBSTRT1	model.meta.subarray.xstart	NIRSpec only
SUBSTRT2	model.meta.subarray.ystart	NIRSpec only
SUBSIZE1	model.meta.subarray.xsize	NIRSpec only
SUBSIZE2	model.meta.subarray.ysize	NIRSpec only

Reference File Format

SUPERBIAS reference files are FITS format, with 3 IMAGE extensions and 1 BINTABLE extension. The FITS primary HDU does not contain a data array. The format and content of the file is as follows:

EXTNAME	XTENSION	NAXIS	Dimensions	Data type
SCI	IMAGE	2	ncols x nrows	float
ERR	IMAGE	2	ncols x nrows	float
DQ	IMAGE	2	ncols x nrows	integer
DQ_DEF	BINTABLE	2	TFIELDS = 4	N/A

The `SCI` array contains the super-bias image of the detector. The `ERR` array contains uncertainties in the super-bias values and the `DQ` array contains data quality flags associated with the super-bias image.

The DQ_DEF extension contains the bit assignments used in the DQ array. It contains the following 4 columns:

TTYPE	TFORM	Description
BIT	integer	The bit number, starting at zero
VALUE	integer	The equivalent base-10 value of BIT
NAME	string	The mnemonic name of the data quality condition
DESCRIPTION	string	A description of the data quality condition

NOTE: For more information on standard bit definitions see: *Data Quality Flags*.

jwst.superbias Package

Classes

SuperBiasStep([name, parent, config_file, ...])	SuperBiasStep: Performs super-bias subtraction by subtracting super-bias reference data from the input science data model.

SuperBiasStep

class jwst.superbias.**SuperBiasStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

SuperBiasStep: Performs super-bias subtraction by subtracting super-bias reference data from the input science data model.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'superbias'

reference_file_types = ['superbias']

spec

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.62 TSO Aperture Photometry

Description

Class
jwst.tso_photometry.TSOPhotometryStep

Alias
tso_photometry

The `tso_photometry` step does aperture photometry with a circular aperture for the target. Background is computed as the mean within a circular annulus. The output is a table (ASCII ecsv format) containing the time at the midpoint of each integration and the photometry values.

Assumptions

This step is intended to be used for aperture photometry with time-series exposures. Only direct images should be used, not spectra.

The target is assumed to have been placed at the aperture reference location, which is stored in the XREF_SCI and YREF_SCI FITS keywords (note that these are 1-indexed values). Hence the step uses those keyword values as the target location within the image.

Algorithm

The Astropy affiliated package `photutils` does the work.

If the input file was *not* averaged over integrations (i.e. a _calints product), and if the file contains an INT_TIMES table extension, the times listed in the output table from this step will be extracted from column 'int_mid_MJD_UTC' of the INT_TIMES extension. Otherwise, the times will be computed from the exposure start time, the integration time, and the number of integrations. In either case, the times are Modified Julian Date, time scale UTC.

The output table contains these fields:

- MJD
- aperture_sum
- aperture_sum_err
- annulus_sum
- annulus_sum_err
- annulus_mean
- annulus_mean_err
- aperture_bkg
- aperture_bkg_err
- net_aperture_sum
- net_aperture_sum_err

Subarrays

If a subarray is used that is so small that the target aperture extends beyond the limits of the subarray, the entire area of the subarray will be used for the target aperture, and no background subtraction will be done. A specific example is SUB64 with NIRCam, using PUPIL = WLP8.

Step Arguments

The `tso_photometry` step has one step-specific argument:

- `--save_catalog`

If `save_catalog` is set to True (the default is False), the output table of times and photometry will be written to an ecsv file with suffix "phot".

Note that when this step is run as part of the *calwebb_tso3* pipeline, the `save_catalog` argument should *not* be set, because the output catalog will always be saved by the pipeline module itself. The `save_catalog` argument is useful only when the `tso_photometry` step is run standalone.

Reference Files

The `tso_photometry` step uses a TSOPHOT reference file.

TSOPHOT Reference File

REFTYPE
TSOPHOT

Data model
`TsoPhotModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.TsoPhotModel.html#jwst.datamodels.Tso

The TSOPHOT reference file contains photometry aperture radii for TSO imaging observation modes.

Reference Selection Keywords for TSOPHOT

CRDS selects appropriate TSOPHOT references based on the following keywords. TSOPHOT is not applicable for instruments not in the table. All keywords used for file selection are *required*.

Instrument	Keywords
MIRI	INSTRUME, EXP_TYPE, TSOVISIT, DATE-OBS, TIME-OBS
NIRCam	INSTRUME, EXP_TYPE, TSOVISIT, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for TSOPHOT

In addition to the standard reference file keywords listed above, the following keywords are *required* in TSOPHOT reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for TSOPHOT*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type
TSOVISIT	model.meta.visit.tsovisit

Reference File Format

TSOPHOT reference files are ASDF format. An object called 'radii' in a TSOPHOT file defines the radii that the step needs. This object is a list of one or more dictionaries. Each such dictionary has four keys: 'pupil', 'radius', 'radius_inner', and 'radius_outer'. The particular one of these dictionaries to use is selected by comparing meta.instrument.pupil with the value corresponding to 'pupil' in each dictionary. If an exact match is found, that dictionary will be used. If no match is found, the first dictionary with 'pupil': 'ANY' will be selected. The radii will be taken from the values of keys 'radius', 'radius_inner', and 'radius_outer'.

jwst.tso_photometry Package

Classes

TSOPhotometryStep([name, parent, ...])	Perform circular aperture photometry on imaging Time Series Observations (TSO).

TSOPhotometryStep

class jwst.tso_photometry.**TSOPhotometryStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

Perform circular aperture photometry on imaging Time Series Observations (TSO).

Parameters
input (str or CubeModel) – Filename for a FITS image or association table, or a CubeModel.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input_data)	This is where real work happens.

Attributes Documentation

class_alias = 'tso_photometry'

reference_file_types = ['tsophot']

spec

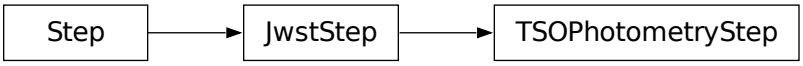
```
save_catalog = boolean(default=False)  # save exposure-level catalog
```

Methods Documentation

process(*input_data*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.63 TweakReg

Description

Class

jwst.tweakreg.TweakRegStep

Alias

tweakreg

Overview

This step creates image catalogs of point-like sources whose centroids are then used to compute corrections to the WCS of the input images such that sky catalogs obtained from the image catalogs using the corrected WCS will align on the sky.

Source Detection

If `meta.tweakreg_catalog` attribute of input data models is a non-empty string and `use_custom_catalogs` is `True` (https://docs.python.org/3/library/constants.html#True), then it will be interpretted as a file name of a user-provided source catalog. The catalog must be in a format automatically recognized by `read()`.

When `meta.tweakreg_catalog` attribute of input data models is `None` (https://docs.python.org/3/library/constants.html#None) or an empty string, then `tweakreg` step will attempt to detect sources in the input images. Stars are detected in the image using the Photutils "daofind" function. Photutils.daofind is an implementation of the DAOFIND (http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?daofind) algorithm (Stetson 1987, PASP 99, 191 (http://adsabs.harvard.edu/abs/1987PASP...99..191S)). It searches images for local density maxima that have a peak amplitude greater than a specified threshold (the threshold is applied to a convolved image) and have a size and shape similar to a defined 2D Gaussian kernel. `photutils.daofind` also provides an estimate of the objects roundness and sharpness, whose lower and upper bounds can be specified.

Custom Source Catalogs

Source detection built-in into the `tweakreg` step can be disabled by providing a file name to a custom source catalog in the `meta.tweakreg_catalog` attribute of input data models. The catalog must be in a format automatically recognized by `read()`. The catalog must contain either `'x'` and `'y'` or `'xcentroid'` and `'ycentroid'` columns which indicate source *image* coordinates (in pixels). Pixel coordinates are 0-indexed. An optional column in the catalog is the `'weight'` column, which when present, will be used in fitting.

For the `tweakreg` step to use user-provided input source catalogs, `use_custom_catalogs` parameter of the `tweakreg` step must be set to `True` (https://docs.python.org/3/library/constants.html#True).

In addition to setting the `meta.tweakreg_catalog` attribute of input data models to the custom catalog file name, the `tweakreg_step` also supports two other ways of supplying custom source catalogs to the step:

1. Adding `tweakreg_catalog` attribute to the `members` of the input ASN table - see `ModelContainer` for more details. Catalog file names are relative to ASN file path.
2. Providing a simple two-column text file, specified via step's parameter `catfile`, that contains input data models' file names in the first column and the file names of the corresponding catalogs in the second column. Catalog file names are relative to `catfile` file path.

Specifying custom source catalogs via either the input ASN file or `catfile`, will update input data models' `meta.tweakreg_catalog` attributes to the catalog file names provided in either in the ASN file or `catfile`.

Note: When custom source catalogs are provided via both `catfile` and ASN file members' attributes, the `catfile` takes precedence and catalogs specified via ASN file are ignored altogether.

Note:

1. Providing a data model file name in the `catfile` and leaving the corresponding source catalog file name empty – same as setting `'tweakreg_catalog'` in the ASN file to an empty string `""` – would set corresponding input data

model's `meta.tweakreg_catalog` attribute to `None` (https://docs.python.org/3/library/constants.html#None). In this case, `tweakreg_step` will automatically generate a source catalog for that data model.

2. If an input data model is not listed in the `catfile` or does not have `'tweakreg_catalog'` attribute provided in the ASN file, then the catalog file name in that model's `meta.tweakreg_catalog` attribute will be used. If `model.meta.tweakreg_catalog` is `None` (https://docs.python.org/3/library/constants.html#None), `tweakreg_step` will automatically generate a source catalog for that data model.

Alignment

The source catalogs for each input image are compared to each other and linear (affine) coordinate transformations that align these catalogs are derived. This fit ensures that all the input images are aligned relative to each other. This step produces a combined source catalog for the entire set of input images as if they were combined into a single mosaic.

If the step parameter `abs_refcat` is set to 'GAIADR3', 'GAIADR2' or 'GAIADR1', an astrometric reference catalog then gets generated by querying a GAIA-based astrometric catalog web service for all astrometrically measured sources in the combined field-of-view of the set of input images. This catalog is generated from the catalogs available through the STScI MAST Catalogs (https://outerspace.stsci.edu/display/MASTDATA/Catalog+Access) and has the ability to account for proper motion to a given epoch. The epoch is computed from the observation date and time of the input data.

The combined source catalog derived in the first step then gets cross-matched and fit to this astrometric reference catalog. The pipeline initially supports fitting to the GAIADR3 catalog, with the option to select the GAIADR2 or GAIADR1 instead. The results of this one fit then gets back-propagated to all the input images to align them all to the astrometric reference frame while maintaining the relative alignment between the images.

For this part of alignment, instead of 'GAIADR1', 'GAIADR2', or 'GAIADR3', users can supply an external reference catalog by providing a path to an existing file. User-supplied catalog must contain `'RA'` and `'DEC'` columns which indicate reference source world coordinates (in degrees). An optional column in the catalog is the `'weight'` column, which when present, will be used in fitting. The catalog must be in a format automatically recognized by `read()`.

Grouping

Images taken at the same time (e.g., NIRCam images from all short-wave detectors) can be aligned together, that is, a single correction can be computed and applied to all these images because any error in telescope pointing will be identical in all these images and it is assumed that the relative positions of (e.g., NIRCam) detectors do not change. Identification of images that belong to the same "exposure" and therefore can be grouped together is based on several attributes described in `ModelContainer`. This grouping is performed automatically in the `tweakreg` step using the `models_grouped` property, which assigns a group ID to each input image model in `meta.group_id`.

However, when detector calibrations are not accurate, alignment of groups of images may fail (or result in poor alignment). In this case, it may be desirable to align each image independently. This can be achieved either by setting the `image_model.meta.group_id` attribute to a unique string or integer value for each image, or by adding the `group_id` attribute to the `members` of the input ASN table - see `ModelContainer` for more details.

Note: Group ID (`group_id`) is used by both `tweakreg` and `skymatch` steps and so modifying it for one step will affect the results in another step. If it is desirable to apply different grouping strategies to the `tweakreg` and `skymatch` steps, one may need to run each step individually and provide a different ASN as input to each step.

WCS Correction

The linear coordinate transformation computed in the previous step is used to define tangent-plane corrections that need to be applied to the GWCS pipeline in order to correct input image WCS. This correction is implemented by inserting a `v2v3corr` frame with tangent plane corrections into the GWCS pipeline of the image's WCS.

Step Arguments

The `tweakreg` step has the following optional arguments:

Source finding parameters:

- `save_catalogs`: A boolean indicating whether or not the catalogs should be written out. This parameter is ignored for input data models whose `meta.tweakreg_catalog` is a non-empty string pointing to a user-supplied source catalog. (Default=`False`)
- `use_custom_catalogs`: A boolean that indicates whether to ignore source catalog in the input data model's `meta.tweakreg_catalog` attribute. If `False` (https://docs.python.org/3/library/constants.html#False), new catalogs will be generated by the `tweakreg` step. (Default=`False`)
- `catalog_format`: A `str` (https://docs.python.org/3/library/stdtypes.html#str) indicating catalog output file format. (Default='ecsv')
- `catfile`: Name of the file with a list of custom user-provided catalogs. (Default='')
- `kernel_fwhm`: A `float` (https://docs.python.org/3/library/functions.html#float) value indicating the Gaussian kernel FWHM in pixels. (Default=2.5)
- `snr_threshold`: A `float` (https://docs.python.org/3/library/functions.html#float) value indicating SNR threshold above the background. (Default=5.0)
- `sharplo`: A `float` (https://docs.python.org/3/library/functions.html#float) value indicating The lower bound on sharpness for object detection. (Default=0.2)
- `sharphi`: A `float` (https://docs.python.org/3/library/functions.html#float) value indicating the upper bound on sharpness for object detection. (Default=1.0)
- `roundlo`: A `float` (https://docs.python.org/3/library/functions.html#float) value indicating the lower bound on roundness for object detection. (Default=-1.0)
- `roundhi`: `float` (https://docs.python.org/3/library/functions.html#float) value indicating the upper bound on roundness for object detection. (Default=1.0)
- `brightest`: A positive `int` (https://docs.python.org/3/library/functions.html#int) value indicating the number of brightest objects to keep. (Default=200)
- `peakmax`: A `float` (https://docs.python.org/3/library/functions.html#float) value used to filter out objects with pixel values >= `peakmax`. (Default=None)
- `bkg_boxsize`: A positive `int` (https://docs.python.org/3/library/functions.html#int) indicating the background mesh box size in pixels. (Default=400)

Optimize alignment order:

- `enforce_user_order`: a boolean value indicating whether or not take the first image as a reference image and then align the rest of the images to that reference image in the order in which input images have been provided or to optimize order in which images are aligned. (Default=`False`)

Reference Catalog parameters:

- `expand_refcat`: A boolean indicating whether or not to expand reference catalog with new sources from other input images that have been already aligned to the reference image. (Default=False)

Object matching parameters:

- `minobj`: A positive `int` (https://docs.python.org/3/library/functions.html#int) indicating minimum number of objects acceptable for matching. (Default=15)
- `searchrad`: A `float` (https://docs.python.org/3/library/functions.html#float) indicating the search radius in arcsec for a match. (Default=2.0)
- `use2dhist`: A boolean indicating whether to use 2D histogram to find initial offset. (Default=True)
- `separation`: Minimum object separation in arcsec. (Default=1.0)
- `tolerance`: Matching tolerance for `xyxymatch` in arcsec. (Default=0.7)
- `xoffset`: Initial guess for X offset in arcsec. (Default=0.0)
- `yoffset`: Initial guess for Y offset in arcsec. (Default=0.0)

Catalog fitting parameters:

- `fitgeometry`: A `str` (https://docs.python.org/3/library/stdtypes.html#str) value indicating the type of affine transformation to be considered when fitting catalogs. Allowed values:
  - `'shift'`: x/y shifts only
  - `'rshift'`: rotation and shifts
  - `'rscale'`: rotation and scale
  - `'general'`: shift, rotation, and scale

  The default value is "rshift".

  Note: Mathematically, alignment of images observed in different tangent planes requires `fitgeometry='general'` in order to fit source catalogs in the different images even if mis-alignment is caused only by a shift or rotation in the tangent plane of one of the images.

  However, under certain circumstances, such as small alignment errors or minimal dithering during observations that keep tangent planes of the images to be aligned almost parallel, then it may be more robust to use a `fitgeometry` setting with fewer degrees of freedom such as `'rshift'`, especially for "ill-conditioned" source catalogs such as catalogs with very few sources, or large errors in source positions, or sources placed along a line or bunched in a corner of the image (not spread across/covering the entire image).

- `nclip`: A non-negative integer number of clipping iterations to use in the fit. (Default = 3)
- `sigma`: A positive `float` (https://docs.python.org/3/library/functions.html#float) indicating the clipping limit, in sigma units, used when performing fit. (Default=3.0)

Absolute Astrometric fitting parameters:

Parameters used for absolute astrometry to a reference catalog.

- `abs_refcat`: String indicating what astrometric catalog should be used. Currently supported options: 'GAIADR1', 'GAIADR2', 'GAIADR3', a path to an existing reference catalog, `None` (https://docs.python.org/3/library/constants.html#None), or ''. See `jwst.tweakreg.tweakreg_step.SINGLE_GROUP_REFCAT` for an up-to-date list of supported built-in reference catalogs.

  When `abs_refcat` is `None` (https://docs.python.org/3/library/constants.html#None) or an empty string, alignment to the absolute astrometry catalog will be turned off. (Default='')

- `abs_minobj`: A positive `int` (https://docs.python.org/3/library/functions.html#int) indicating minimum number of objects acceptable for matching. (Default=15)

- `abs_searchrad`: A `float` (https://docs.python.org/3/library/functions.html#float) indicating the search radius in arcsec for a match. It is recommended that a value larger than `searchrad` be used for this parameter (e.g. 3 times larger) (Default=6.0)
- `abs_use2dhist`: A boolean indicating whether to use 2D histogram to find initial offset. It is strongly recommended setting this parameter to `True` (https://docs.python.org/3/library/constants.html#True). Otherwise the initial guess for the offsets will be set to zero (Default=True)
- `abs_separation`: Minimum object separation in arcsec. It is recommended that a value smaller than `separation` be used for this parameter (e.g. 10 times smaller) (Default=0.1)
- `abs_tolerance`: Matching tolerance for `xyxymatch` in arcsec. (Default=0.7)
- `abs_fitgeometry`: A `str` (https://docs.python.org/3/library/stdtypes.html#str) value indicating the type of affine transformation to be considered when fitting catalogs. Allowed values:
  - `'shift'`: x/y shifts only
  - `'rshift'`: rotation and shifts
  - `'rscale'`: rotation and scale
  - `'general'`: shift, rotation, and scale

  The default value is "rshift". Note that the same conditions/restrictions that apply to `fitgeometry` also apply to `abs_fitgeometry`.
- `abs_nclip`: A non-negative integer number of clipping iterations to use in the fit. (Default = 3)
- `abs_sigma`: A positive `float` (https://docs.python.org/3/library/functions.html#float) indicating the clipping limit, in sigma units, used when performing fit. (Default=3.0)
- `save_abs_catalog`: A boolean specifying whether or not to write out the astrometric catalog used for the fit as a separate product. (Default=False)

Further Documentation

The underlying algorithms as well as formats of source catalogs are described in more detail at

https://tweakwcs.readthedocs.io/en/latest/

Reference Files

The `tweakreg` step uses the PARS-TWEAKREGSTEP parameter reference file.

PARS-TWEAKREGSTEP Parameter Reference File

REFTYPE
PARS-TWEAKREGSTEP

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-tweakregstep references based on the following keywords.

Instrument	Keywords
FGS	EXP_TYPE
MIRI	EXP_TYPE, FILTER
NIRCAM	EXP_TYPE, FILTER, PUPIL
NIRISS	EXP_TYPE, FILTER, PUPIL

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Also See:

tweakreg_catalog

The `tweakreg_catalog` module provides functions for generating catalogs of sources from images.

jwst.tweakreg.tweakreg_catalog Module

Functions

make_tweakreg_catalog(model, kernel_fwhm, ...)	Create a catalog of point-line sources to be used for image alignment in tweakreg.

make_tweakreg_catalog

jwst.tweakreg.tweakreg_catalog.**make_tweakreg_catalog**(*model*, *kernel_fwhm*, *snr_threshold*, *sharplo=0.2*, *sharphi=1.0*, *roundlo=-1.0*, *roundhi=1.0*, *brightest=None*, *peakmax=None*, *bkg_boxsize=400*)

Create a catalog of point-line sources to be used for image alignment in tweakreg.

Parameters

- **model** (ImageModel) – The input ImageModel of a single image. The input image is assumed to be background subtracted.
- **kernel_fwhm** (*float* (https://docs.python.org/3/library/functions.html#float)) – The full-width at half-maximum (FWHM) of the 2D Gaussian kernel used to filter the image before thresholding. Filtering the image will smooth the noise and maximize detectability of objects with a shape similar to the kernel.
- **snr_threshold** (*float* (https://docs.python.org/3/library/functions.html#float)) – The signal-to-noise ratio per pixel above the background for which to consider a pixel as possibly being part of a source.
- **sharplo** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The lower bound on sharpness for object detection.
- **sharphi** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The upper bound on sharpness for object detection.
- **roundlo** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The lower bound on roundness for object detection.
- **roundhi** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The upper bound on roundness for object detection.
- **brightest** (*int* (https://docs.python.org/3/library/functions.html#int), *None*, *optional*) – Number of brightest objects to keep after sorting the full object list. If brightest is set to None (https://docs.python.org/3/library/constants.html#None), all objects will be selected.
- **peakmax** (*float* (https://docs.python.org/3/library/functions.html#float), *None*, *optional*) – Maximum peak pixel value in an object. Only objects whose peak pixel values are *strictly smaller* than peakmax will be selected. This may be used to exclude saturated sources. By default, when peakmax is set to None (https://docs.python.org/3/library/constants.html#None), all objects will be selected.

  Warning: DAOStarFinder automatically excludes objects whose peak pixel values are negative. Therefore, setting peakmax to a non-positive value would result in exclusion of all objects.

- **bkg_boxsize** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The background mesh box size in pixels.

Returns

catalog – An astropy Table containing the source catalog.

Return type

Table

tweakreg_step

The `tweakreg_step` function (class name `TweakRegStep`) is the top-level function used to call the "tweakreg" operation from the JWST calibration pipeline.

jwst.tweakreg.tweakreg_step Module

JWST pipeline step for image alignment.

Authors
Mihai Cara

Classes

TweakRegStep([name, parent, config_file, ...])	TweakRegStep: Image alignment based on catalogs of sources detected in input images.

TweakRegStep

class jwst.tweakreg.tweakreg_step.**TweakRegStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

TweakRegStep: Image alignment based on catalogs of sources detected in input images.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'tweakreg'

reference_file_types = []

spec

```
save_catalogs = boolean(default=False) # Write out catalogs?
use_custom_catalogs = boolean(default=False) # Use custom user-provided␣
↪catalogs?
catalog_format = string(default='ecsv') # Catalog output file format
catfile = string(default='') # Name of the file with a list of custom user-
↪provided catalogs
kernel_fwhm = float(default=2.5) # Gaussian kernel FWHM in pixels
snr_threshold = float(default=10.0) # SNR threshold above the bkg
sharplo = float(default=0.2) # The lower bound on sharpness for object␣
↪detection.
sharphi = float(default=1.0) # The upper bound on sharpness for object␣
↪detection.
roundlo = float(default=-1.0) # The lower bound on roundness for object␣
↪detection.
roundhi = float(default=1.0) # The upper bound on roundness for object␣
↪detection.
brightest = integer(default=200) # Keep top ``brightest`` objects
peakmax = float(default=None) # Filter out objects with pixel values >=␣
↪``peakmax``
bkg_boxsize = integer(default=400) # The background mesh box size in pixels.
enforce_user_order = boolean(default=False) # Align images in user specified␣
↪order?
expand_refcat = boolean(default=False) # Expand reference catalog with new␣
↪sources?
minobj = integer(default=15) # Minimum number of objects acceptable for matching
searchrad = float(default=2.0) # The search radius in arcsec for a match
use2dhist = boolean(default=True) # Use 2d histogram to find initial offset?
separation = float(default=1.0) # Minimum object separation in arcsec
tolerance = float(default=0.7) # Matching tolerance for xyxymatch in arcsec
xoffset = float(default=0.0), # Initial guess for X offset in arcsec
yoffset = float(default=0.0) # Initial guess for Y offset in arcsec
fitgeometry = option('shift', 'rshift', 'rscale', 'general', default='rshift')
↪# Fitting geometry
nclip = integer(min=0, default=3) # Number of clipping iterations in fit
sigma = float(min=0.0, default=3.0) # Clipping limit in sigma units
abs_refcat = string(default='')  # Catalog file name or one of: "'GAIADR3'", "
↪'GAIADR2'", or "'GAIADR1'", or None, or "
save_abs_catalog = boolean(default=False)  # Write out used absolute␣
↪astrometric reference catalog as a separate product
```

(continues on next page)

(continued from previous page)

```
abs_minobj = integer(default=15) # Minimum number of objects acceptable for␣
↪matching when performing absolute astrometry
abs_searchrad = float(default=6.0) # The search radius in arcsec for a match␣
↪when performing absolute astrometry
# We encourage setting this parameter to True. Otherwise, xoffset and yoffset␣
↪will be set to zero.
abs_use2dhist = boolean(default=True) # Use 2D histogram to find initial offset␣
↪when performing absolute astrometry?
abs_separation = float(default=0.1) # Minimum object separation in arcsec when␣
↪performing absolute astrometry
abs_tolerance = float(default=0.7) # Matching tolerance for xyxymatch in arcsec␣
↪when performing absolute astrometry
# Fitting geometry when performing absolute astrometry
abs_fitgeometry = option('shift', 'rshift', 'rscale', 'general', default='rshift
↪')
abs_nclip = integer(min=0, default=3) # Number of clipping iterations in fit␣
↪when performing absolute astrometry
abs_sigma = float(min=0.0, default=3.0) # Clipping limit in sigma units when␣
↪performing absolute astrometry
output_use_model = boolean(default=True)  # When saving use `DataModel.meta.
↪filename`
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

Utility Functions

Currently, the `utils` module provides helpful functions for manually applying corrections to an imaging WCS.

jwst.tweakreg.utils Module

Functions

adjust_wcs(wcs[, delta_ra, delta_dec, ...])	Apply corrections to an imaging WCS of 'cal' data models.
transfer_wcs_correction(to_image, from_image)	Applies the same *total* WCS correction that was applied by `tweakreg` (!) to the WCS in the `from_image` data model to the WCS of the `to_image` data model.

adjust_wcs

jwst.tweakreg.utils.**adjust_wcs**(*wcs*, *delta_ra=0.0*, *delta_dec=0.0*, *delta_roll=0.0*, *scale_factor=1.0*)

Apply corrections to an imaging WCS of 'cal' data models.

> **Warning:** This function is not designed to handle neither FITS WCS nor GWCS of resampled images. It is designed specifically for GWCS of calibrated imaging data models that can be used as input to Stage 3 of the JWST pipeline (with suffixes '_cal', '_tweakreg', '_skymatch').

> **Warning:** This function modifies the WCS of calibrated imaging data models in a way that is NOT compatible with `tweakreg`: once a WCS was modified using `adjust_wcs()`, the corresponding imaging data model (whose WCS was modified) no longer be aligned using the `tweakreg` step.

Parameters

- **wcs** (`gwcs.WCS`) – WCS object to be adjusted. Must be an imaging JWST WCS of a calibrated data model.
- **delta_ra** (*float* (https://docs.python.org/3/library/functions.html#float), *astropy.units.Quantity*, *optional*) – Additional rotation (in degrees if units not provided) to be applied along the longitude direction.
- **delta_dec** (*float* (https://docs.python.org/3/library/functions.html#float), *astropy.units.Quantity*, *optional*) – Additional rotation (in degrees if units not provided) to be applied along the latitude direction.
- **delta_roll** (*float* (https://docs.python.org/3/library/functions.html#float), *astropy.units.Quantity*, *optional*) – Additional rotation (in degrees if units not provided) to be applied to the telescope roll angle (rotation about V1 axis).
- **scale_factor** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – A multiplicative scale factor to be applied to the current scale (if any) in the WCS. If input `wcs` does not have a scale factor applied, it is assumed to be 1. The scale factor is applied in a tangent plane perpendicular to the V1 axis of the telescope.

Returns

wcs – Adjusted WCS object.

Return type

`gwcs.WCS`

transfer_wcs_correction

jwst.tweakreg.utils.**transfer_wcs_correction**(*to_image*, *from_image*, *matrix=None*, *shift=None*)

Applies the same *total* WCS correction that was applied by `tweakreg` (!) to the WCS in the `from_image` data model to the WCS of the `to_image` data model. In some ways this function is analogous function to the `tweakback` function for HST available in the drizzlepac package (https://github.com/spacetelescope/drizzlepac).

One fundamental difference between this function and `tweakback` is that JWST data models do not keep a history of data's WCS via alternative WCS as it is done in HST data and so it is impossible to select and apply only one particular WCS correction if there were multiple corrections previously applied to a WCS. The tangent-plane correction in JWST WCS is cumulative/total correction. If you would like to apply a specific/custom correction, you can do that via `matrix` and `shift` arguments which is defined in the reference tangent plane provided by the `from_image`'s WCS.

When providing your own corrections via `matrix` and `shift` arguments, this function is similar to the *adjust_wcs()* function but provides an alternative way of specifying corrections via affine transformations in a reference tangent plane.

Warning: Upon return, if the `to_image` argument is an `ImageModel` it will be modified with an updated `ImageModel.meta.wcs` WCS model. If `to_image` argument is a file name of an `ImageModel`, that model will be read in, its WCS will be updated, and the updated model will be written out to the same file. BACKUP the file in `to_image` argument before calling this function.

Warning: This function does not support input data models whose WCS were modified by *adjust_wcs()*. Only WCS corrections computed by either the `tweakreg` step or by `tweakwcs` package are supported.

Parameters

- **to_image** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *ImageModel*) – Image model to which the correction should be applied/transferred to.

  Warning: If it is a string file name then, upon return, this file will be **overwritten** with a data model with an updated WCS.

- **from_image** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *ImageModel*, *gwcs.wcs.WCS* (https://gwcs.readthedocs.io/en/stable/api/gwcs.wcs.WCS.html#gwcs.wcs.WCS)) – A data model whose WCS was previously corrected. This data model plays two roles: 1) it is the reference WCS which provides a tangent plane in which corrections have been defined, and 2) it provides WCS corrections to be applied to `to_image`'s WCS.

  If the WCS of the `from_image` data model does not contain corrections, then *both* `matrix` *and* `shift` arguments *must be supplied.*

- **matrix** (*2D list, 2D numpy.ndarray, None, optional*) – A 2D matrix part of an affine transformation defined in the tangent plane derived from the `from_image`'s WCS.

  Note: When provided, `shift` argument *must also be provided* in which case `matrix` and `shift` arguments override the correction (if present) from the `from_file`'s WCS.

- **shift** (*list* (https://docs.python.org/3/library/stdtypes.html#list), *numpy.ndarray* (https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray), *None, optional*) – A list of length 2 representing the translational part of an affine transformation (in arcsec) defined in the tangent plane derived from the `from_image`'s WCS.

Note: When provided, `matrix` argument *must also be provided* in which case `matrix` and `shift` arguments override the correction (if present) from the `from_file`'s WCS.

Returns

- Upon return, if the `to_image` argument is an `ImageModel` it will be
- modified with an updated `ImageModel.meta.wcs` WCS model.
- If `to_image` argument is a file name of an `ImageModel`, that
- *model will be read in, its WCS will be updated, and the updated model*
- will be written to the same file. BACKUP the file in `to_image`
- *argument before calling this function.*

astrometric_utils

The `astrometric_utils` module provides functions for generating astrometric catalogs of sources for the field-of-view covered by a set of images.

jwst.tweakreg.astrometric_utils Module

Functions

compute_radius(wcs)	Compute the radius from the center to the furthest edge of the WCS.
create_astrometric_catalog(input_models[, ...])	Create an astrometric catalog that covers the inputs' field-of-view.
get_catalog(ra, dec[, epoch, sr, catalog])	Extract catalog from VO web service.

compute_radius

`jwst.tweakreg.astrometric_utils.`**`compute_radius`**`(`*`wcs`*`)`

Compute the radius from the center to the furthest edge of the WCS.

create_astrometric_catalog

jwst.tweakreg.astrometric_utils.**create_astrometric_catalog**(*input_models*, *catalog='GAIADR3'*, *output='ref_cat.ecsv'*, *gaia_only=False*, *table_format='ascii.ecsv'*, *existing_wcs=None*, *num_sources=None*, *epoch=None*)

Create an astrometric catalog that covers the inputs' field-of-view.

Parameters

- **input_models** (list of `JwstDataModel`) – Each datamodel must have a ~gwcs.WCS object.
- **catalog** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – Name of catalog to extract astrometric positions for sources in the input images' field-of-view. Default: GAIADR3. Options available are documented on the catalog web page.
- **output** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – Filename to give to the astrometric catalog read in from the master catalog web service. If None, no file will be written out.
- **gaia_only** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Specify whether or not to only use sources from GAIA in output catalog
- **existing_wcs** (*model*) – existing WCS object specified by the user as generated by `resample.resample_utils.make_output_wcs`
- **num_sources** (*int* (https://docs.python.org/3/library/functions.html#int)) – Maximum number of brightest/faintest sources to return in catalog. If `num_sources` is negative, return that number of the faintest sources. By default, all sources are returned.
- **epoch** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Reference epoch used to update the coordinates for proper motion (in decimal year). If `None` (https://docs.python.org/3/library/constants.html#None) then the epoch is obtained from the metadata.

Notes

This function will point to astrometric catalog web service defined through the use of the ASTROMETRIC_CATALOG_URL environment variable.

Returns

ref_table – Astropy Table object of the catalog

Return type

`Table`

get_catalog

jwst.tweakreg.astrometric_utils.**get_catalog**(*ra*, *dec*, *epoch=2016.0*, *sr=0.1*, *catalog='GAIADR3'*)

Extract catalog from VO web service.

Parameters

- **ra** (*float* (https://docs.python.org/3/library/functions.html#float)) – Right Ascension (RA) of center of field-of-view (in decimal degrees)

- **dec** (*float* (https://docs.python.org/3/library/functions.html#float)) – Declination (Dec) of center of field-of-view (in decimal degrees)
- **epoch** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Reference epoch used to update the coordinates for proper motion (in decimal year). Default: 2016.0
- **sr** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Search radius (in decimal degrees) from field-of-view center to use for sources from catalog. Default: 0.1 degrees
- **catalog** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – Name of catalog to query, as defined by web-service. Default: 'GAIADR3'

Returns
csv – CSV object of returned sources with all columns as provided by catalog

Return type
CSV object

Variables

TIMEOUT	Primary function for creating an astrometric reference catalog.

TIMEOUT

jwst.tweakreg.astrometric_utils.**TIMEOUT = 30.0**
Primary function for creating an astrometric reference catalog.

jwst.tweakreg Package

This package provides support for image alignment.

Classes

TweakRegStep([name, parent, config_file, ...])	TweakRegStep: Image alignment based on catalogs of sources detected in input images.

TweakRegStep

class jwst.tweakreg.**TweakRegStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

TweakRegStep: Image alignment based on catalogs of sources detected in input images.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'tweakreg'

reference_file_types = []

spec

```
save_catalogs = boolean(default=False) # Write out catalogs?
use_custom_catalogs = boolean(default=False) # Use custom user-provided␣
↪catalogs?
catalog_format = string(default='ecsv') # Catalog output file format
catfile = string(default='') # Name of the file with a list of custom user-
↪provided catalogs
kernel_fwhm = float(default=2.5) # Gaussian kernel FWHM in pixels
snr_threshold = float(default=10.0) # SNR threshold above the bkg
sharplo = float(default=0.2) # The lower bound on sharpness for object␣
↪detection.
sharphi = float(default=1.0) # The upper bound on sharpness for object␣
↪detection.
roundlo = float(default=-1.0) # The lower bound on roundness for object␣
↪detection.
roundhi = float(default=1.0) # The upper bound on roundness for object␣
↪detection.
```

(continues on next page)

(continued from previous page)

```
brightest = integer(default=200) # Keep top ``brightest`` objects
peakmax = float(default=None) # Filter out objects with pixel values >=␣
↪``peakmax``
bkg_boxsize = integer(default=400) # The background mesh box size in pixels.
enforce_user_order = boolean(default=False) # Align images in user specified␣
↪order?
expand_refcat = boolean(default=False) # Expand reference catalog with new␣
↪sources?
minobj = integer(default=15) # Minimum number of objects acceptable for matching
searchrad = float(default=2.0) # The search radius in arcsec for a match
use2dhist = boolean(default=True) # Use 2d histogram to find initial offset?
separation = float(default=1.0) # Minimum object separation in arcsec
tolerance = float(default=0.7) # Matching tolerance for xyxymatch in arcsec
xoffset = float(default=0.0), # Initial guess for X offset in arcsec
yoffset = float(default=0.0) # Initial guess for Y offset in arcsec
fitgeometry = option('shift', 'rshift', 'rscale', 'general', default='rshift')
↪# Fitting geometry
nclip = integer(min=0, default=3) # Number of clipping iterations in fit
sigma = float(min=0.0, default=3.0) # Clipping limit in sigma units
abs_refcat = string(default='')  # Catalog file name or one of: "'GAIADR3'", "
↪'GAIADR2'", or "'GAIADR1'", or None, or "
save_abs_catalog = boolean(default=False)  # Write out used absolute␣
↪astrometric reference catalog as a separate product
abs_minobj = integer(default=15) # Minimum number of objects acceptable for␣
↪matching when performing absolute astrometry
abs_searchrad = float(default=6.0) # The search radius in arcsec for a match␣
↪when performing absolute astrometry
# We encourage setting this parameter to True. Otherwise, xoffset and yoffset␣
↪will be set to zero.
abs_use2dhist = boolean(default=True) # Use 2D histogram to find initial offset␣
↪when performing absolute astrometry?
abs_separation = float(default=0.1) # Minimum object separation in arcsec when␣
↪performing absolute astrometry
abs_tolerance = float(default=0.7) # Matching tolerance for xyxymatch in arcsec␣
↪when performing absolute astrometry
# Fitting geometry when performing absolute astrometry
abs_fitgeometry = option('shift', 'rshift', 'rscale', 'general', default='rshift
↪')
abs_nclip = integer(min=0, default=3) # Number of clipping iterations in fit␣
↪when performing absolute astrometry
abs_sigma = float(min=0.0, default=3.0) # Clipping limit in sigma units when␣
↪performing absolute astrometry
output_use_model = boolean(default=True)  # When saving use `DataModel.meta.
↪filename`
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.64 Wavelength Correction

Description

Class

jwst.wavecorr.WavecorrStep

Alias

wavecorr

The wavelength correction (`wavecorr`) step in the *calwebb_spec2* pipeline updates the wavelength assignments for NIRSpec fixed-slit (FS) and MOS point sources that are known to be off center (in the dispersion direction) in their slit.

NIRSpec MOS

For NIRSpec MOS exposures (EXP_TYPE="NRS_MSASPEC"), wavelength assignments created during *extract_2d* are based on a source that's perfectly centered in a slitlet. Most sources, however, are not centered in every slitlet in a real observation. The MSA meta data assigned to each slitlet in the *extract_2d* step includes estimates of the source x (dispersion) and y (cross-dispersion) location within the slitlet. These are recorded in the "SRCXPOS" and "SRCYPOS" keywords in the SCI extension header of each slitlet in a FITS product.

The `wavecorr` step loops over all slit instances in the input science product and applies a wavelength correction to slits that contain a point source. The point source determination is based on the value of the "SRCTYPE" keyword populated for each slit by the *srctype* step. The computation of the correction is based on the "SRCXPOS" value. A value of 0.0 indicates a perfectly centered source, and ranges from -0.5 to +0.5 for sources at the extreme edges of a slit. The computation uses calibration data from the `WAVECORR` reference file. The correction is computed as a 2-D grid of wavelength offsets, which is applied to the original 2-D grid of wavelengths associated with each slit.

NIRSpec Fixed Slit (FS)

Fixed slit data do not have an *a priori* estimate of the source location within a given slit, so the estimated source location is computed by the `wavecorr` step. It uses the target coordinates in conjunction with the aperture reference point in V2/V3 space to estimate the fractional location of the source within the given slit. Note that this computation can only be performed for the primary slit in the exposure, which is given in the "FXD_SLIT" keyword. The positions of sources in any additional slits cannot be estimated and therefore the wavelength correction is only applied to the primary slit.

The estimated position of the source within the primary slit (in the dispersion direction) is then used in the same manner as described above for MOS slitlets to compute offsets to be added to the nominal wavelength grid for the primary slit.

Upon successful completion of the step, the status keyword "S_WAVCOR" is set to "COMPLETE".

Step Arguments

The Wavecorr step has no step-specific arguments.

Reference Files

The `wavecorr` step uses the WAVECORR reference file, which only applies to NIRSpec fixed-slit (FS) and MOS exposures.

WAVECORR Reference File

REFTYPE
WAVECORR

Data model
`WaveCorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.WaveCorrModel.html#jwst.datamodels.

The WAVECORR reference file contains pixel offset values as a function of wavelength and source offset within a NIRSpec slit. It is used when applying the NIRSpec wavelength zero-point correction to fixed-slit (EXP_TYPE="NRS_FIXEDSLIT"), bright object TSO (EXP_TYPE="NRS_BRIGHTOBJ"), and MSA/MOS spectra (EXP_TYPE="NRS_MSASPEC"). This is an optional correction that is turned on by default. It can be turned off by specifying `apply_wavecorr=False` when running the step.

Reference Selection Keywords for WAVECORR

CRDS selects appropriate WAVECORR references based on the following keywords. WAVECORR is not applicable for instruments not in the table. Non-standard keywords used for file selection are *required*.

Instrument	Keywords
NIRSpec	INSTRUME, EXP_TYPE, DATE-OBS, TIME-OBS

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

Type Specific Keywords for WAVECORR

In addition to the standard reference file keywords listed above, the following keywords are *required* in WAVECORR reference files, because they are used as CRDS selectors (see *Reference Selection Keywords for WAVECORR*):

Keyword	Data Model Name
EXP_TYPE	model.meta.exposure.type

Reference File Format

WAVECORR reference files are in ASDF format, with the format and contents specified by the `WaveCorrModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.WaveCorrModel.html#jwst.datamodels.WaveCorrModel) data model schema.

jwst.wavecorr Package

Classes

WavecorrStep([name, parent, config_file, ...])	This step applies wavelength offsets to off-center NIRSpec sources.

WavecorrStep

class jwst.wavecorr.**WavecorrStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: JwstStep

This step applies wavelength offsets to off-center NIRSpec sources.

Create a Step instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
reference_file_types	
spec	

Methods Summary

process(step_input)	This is where real work happens.

Attributes Documentation

class_alias = 'wavecorr'

reference_file_types = ['wavecorr']

spec

Methods Documentation

process(*step_input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

15.1.65 WFS Combine

Description

The `wfs_combine` step combines a pair of dithered Wavefront Sensing and Control (WFS&C) images. The input images are aligned with one another and then combined using a pixel replacement technique, described in detail below. The images are aligned to only the nearest integer pixel in each direction. No sub-pixel resampling is done.

Due to the WFS dither patterns oscillating between two locations, the first image of the pair will oscillate between the two dither locations. Because the WSS software works in pixel space, we need to change which input image is "image 1" to get star to have the same pixel location in the output image. When the input parameter "flip_dithers" is set to True (the default) and the x offset between image 1 and image 2 is negative, the two images will be switched before any processing is performed.

Algorithm

Creation of the output combined image is a three-step process: first the offsets between the images are computed using the World Coordinate System, then the offsets are used to shift image 2 to be in alignment with image 1, and finally the aligned data from the two images are combined.

Computing Offsets

The WCS transforms of each image are used to compute the RA/Dec values for the center pixel in image 1, and then the pixel indexes of those RA/Dec values are computed in image 2. The difference in the pixel indexes, rounded to the nearest whole pixel, is used as the nominal offsets in the x/y image axes.

If the optional argument "–do_refine" is set to `True`, the nominal offsets are empirically refined using a cross-correlation technique. The steps in the refinement are as follows:

1. Create a smoothed version of image 1 using a Gaussian kernel.
2. Find the approximate centroid of the source in image 1 by computing the mean pixel coordinates, separately in the x and y axes, of all pixel values that are above 50% of the peak signal in the smoothed image.

3. Create subarrays from image 1 and 2 centered on the computed source centroid.
4. Create the cross-correlation image of the two input images.
5. **Find the peak intensity of the cross-correlation image and use this to determine the** refined offset.
6. **Use the find the difference between the cross-correlation pixel offsets and the WCS offsets.** Add these deltas to the nominal offsets computed from the WCS info to form the refined offsets.

Creating the Combined Image

The image 2 data are shifted using the pixel offsets computed above, in order to align it with image 1. For each pixel location in image 1, the output combined image is populated using the following logic:

1. If the pixel values in both image 1 and 2 are good, i.e. DQ=0, the output SCI and ERR image values are the average of the input SCI and ERR values, respectively, and the output DQ is set to 0.
2. If the image 1 pixel is bad (DQ>0) and the image 2 pixel is good, the output SCI and ERR image values are copied from image 2, and the output DQ is set to 0.
3. If the image 1 pixel is good (DQ=0) and the image 2 pixel is bad, the output SCI and ERR image values are copied from image 1, and the output DQ is set to 0.
4. If both image 1 and 2 pixels are bad (DQ>0), the output SCI and ERR image values are set to 0 and the output DQ contains the combination of input DQ values, as well as the "DO_NOT_USE" flag.

Upon successful completion of this step, the status keyword S_WFSCOM will be set to "COMPLETE" in the output image header.

Inputs

2D calibrated images

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_cal

The input to `wfs_combine` is a pair of calibrated ("_cal") exposures, specified via an ASN file. The ASN file may contain a list of several combined products to be created, in which case the step will loop over each set of inputs, creating a combined output for each pair.

Outputs

2D combined image

Data model
`ImageModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.ImageModel.html#jwst.datamodels.ImageM

File suffix
_wfscmb

The output is the combined image, using the product type suffix "_wfscmb."

Step Arguments

The `wfs_combine` step has one step-specific argument:

```
--do_refine  boolean  default=False
```

If set to `True`, the nominal image offsets computed from the WCS information are refined using image cross-correlation. See the algorithm description section for details.

–flip_dithers boolean default=True

When set to True the output star in the combined image from the pairs of WFS images will always be at the same pixel location.

–psf_size float default=100

The largest PSF size in pixels to use for the alignment. This is only used when do_refine==True.

–blur_size flost default=10

The smoothing that is applied for the initial centroiding. his is only used when do_refine==True.

–n_size int default=2

This controls the size of the box used to interpolate in the input images. Should never need to be changed.

Reference File

The `wfs_combine` step does not use any reference files.

jwst.wfs_combine Package

Classes

WfsCombineStep([name, parent, config_file, ...])	This step combines pairs of dithered PSF images

WfsCombineStep

class jwst.wfs_combine.**WfsCombineStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

This step combines pairs of dithered PSF images

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance, optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.

- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(input_table)	This is where real work happens.

Attributes Documentation

class_alias = 'calwebb_wfs-image3'

spec

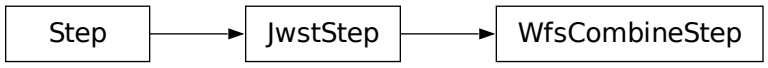
```
do_refine = boolean(default=False)
flip_dithers = boolean(default=True) # change the sign and switch order of␣
↪images when x offset is negative
psf_size = integer(default=100)
blur_size = integer(default=10)
n_size = integer(default=2)
suffix = string(default="wfscmb")
```

Methods Documentation

process(*input_table*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.66 WFSS Contamination Correction

Description

Class
jwst.wfss_contam.WfssContamStep

Alias
wfss_contam

The Wide Field Slitless Spectroscopy (WFSS) contamination correction (`wfss_contam`) step is applied to grism exposures in an attempt to correct effects due to overlapping spectral traces, which often happens in observations of crowded fields. It is to be applied to individual grism exposures in the latter stages of the *calwebb_spec2* pipeline.

Briefly, source fluxes from a direct image of the field are used to simulate grism spectra for each source. Each source spectrum is then corrected for contamination by subtracting the simulated spectra of nearby sources. Details of the procedures and all input/output products are given in the following sections.

Inputs

The method utilized to perform the correction requires several input data products, including:

1) The grism data to be corrected. The step is applied near the end of the *calwebb_spec2* pipeline, after the application of the *extract_2d* and *srctype* steps, but before the *photom* step. Thus individual 2D cutouts exist for each identified source in the grism image, and the data are still in units of countrate.

2) The resampled direct image (*i2d* product) of the field, usually obtained from the same WFSS observation as the grism image. The name of the direct image to use is retrieved from the "DIRIMAGE" keyword in the input grism image, which should've been populated at the beginning of the *calwebb_spec2* pipeline from an entry in the "spec2" input ASN file.

3) The segmentation map (*segm* product) created from the direct image during *calwebb_image3* processing. The name of the segmentation map to use is retrieved from the "SEGMFILE" keyword in the input grism image, which should've been populated at the beginning of the *calwebb_spec2* pipeline from an entry in the "spec2" input ASN file.

The Method

Here we describe the steps used to perform the contamination correction.

1) First, a full-frame intermediate image, matching the size and shape of the grism image to be corrected, is created and populated with simulated spectra of all known sources in the field. The simulated spectra are created as follows:

   a) The segmentation (*segm*) file is searched for pixels with non-zero values and lists of pixels belonging to each source are created.

   b) The fluxes of each pixel in the lists are loaded from the direct image (*i2d*), creating a list of per-pixel flux values for each source.

   c) A list of wavelength values is created for each source, which will be used to create the simulated spectra. The wavelength values span the range given by minimum and maximum wavelengths read from the WAVELENGTHRANGE reference file and are order-dependent.

   d) The direct image pixel locations and wavelengths for each source are transformed into dispersed pixel locations within the grism image using the WCS transforms of the input grism image.

e) The flux of each direct image pixel belonging to each source is "dispersed" into the list of grism image pixel locations, thus creating a simulated spectrum.

f) The initial simulated spectra are in flux-calibrated units, so each spectrum is divided by the sensitivity curve from the PHOTOM reference file, to convert the simulated spectra to units of countrates, thus matching the units of the observed grism data.

g) The simulated spectrum for each source is stored in the full-frame image.

h) Steps c-g are repeated for all spectral orders defined in the WAVELENGTHRANGE reference file.

2) 2D cutouts are created from the full-frame simulated grism image, matching the cutouts of each source in the input grism data.

3) For each source cutout, the simulated spectrum of the primary source is removed from the simulated cutout, leaving only the simulated spectra of any nearby contaminating sources.

4) The simulated contamination cutout is subtracted from the observed source cutout, thereby removing the signal from contaminating spectra.

Outputs

There is one primary output and two optional outputs from the step:

1) The primary output is the contamination-corrected grism data, in the form of a `MultiSlitModel` (https://stdatamodels.readthedocs.io/en/latest/api/jwst.datamodels.MultiSlitModel.html#jwst.datamodels.MultiSlitModel) data model. In the *calwebb_spec2* pipeline flow, this data model is passed along to the *photom* step for further processing.

2) If the step argument `--save_simulated_image` is set to `True` (https://docs.python.org/3/library/constants.html#True), the full-frame image containing all simulated spectra (the result of step 1 above) is saved to a file. See *Step Arguments*.

3) If the step argument `--save_contam_images` is set to `True` (https://docs.python.org/3/library/constants.html#True), the simulated contamination cutouts (the result of step 3 above) are saved to a file. See *Step Arguments*.

Step Arguments

The `wfss_contam` step uses the following optional arguments.

`--save_simulated_image`
A boolean indicating whether the full-frame simulated grism image containing all simulated spectra within the field-of-view should be saved to a file. The file name uses a product type suffix of "simul". Defaults to `False`.

`--save_contam_images`
A boolean indicating whether the estimated contamination images for each source cutout should be saved to a file. The file name uses a product type suffix of "contam". The resulting file has one SCI extension for each source contained in the input grism image. Defaults to `False`.

`--maximum_cores`
The fraction of available cores that will be used for multi-processing in this step. The default value is 'none' which does not use multi-processing. The other options are 'quarter', 'half', and 'all'. Note that these fractions refer to the total available cores and on most CPUs these include physical and virtual cores.

Reference Files

The `wfss_contam` step uses the WAVELENGTHRANGE reference file, which provides minimum and maximum wavelengths for each spectral order, and the PHOTOM reference file, which provides the sensitivity curve for each grism order.

WAVELENGTHRANGE

PHOTOM

jwst.wfss_contam Package

Classes

WfssContamStep([name, parent, config_file, ...])	This Step performs contamination correction of WFSS spectra.

WfssContamStep

class jwst.wfss_contam.**WfssContamStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

This Step performs contamination correction of WFSS spectra.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input_model, *args, **kwargs)	This is where real work happens.

Attributes Documentation

class_alias = 'wfss_contam'

reference_file_types = ['photom', 'wavelengthrange']

spec

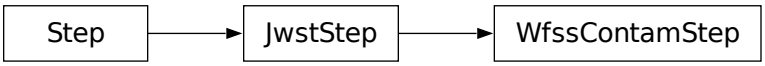
```
save_simulated_image = boolean(default=False)  # Save full-frame simulated image
save_contam_images = boolean(default=False)  # Save source contam estimates
maximum_cores = option('none', 'quarter', 'half', 'all', default='none')
skip = boolean(default=True)
```

Methods Documentation

process(*input_model*, **args*, ***kwargs*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram



15.1.67 White Light Curve Generation

Description

Class
jwst.white_light.WhiteLightStep

Alias
white_light

Overview

The white_light step sums the spectroscopic flux over all wavelengths in each integration of a multi-integration extracted spectrum product to produce an integrated ("white") flux as a function of time for the target. This is to be applied to the _x1dints product in a spectroscopic Time-Series Observation (TSO), as part of the *calwebb_tso3* pipeline. Minimum and maximum wavelengths may be provided to limit the summation to specified wavelength bounds, with limits inclusive.

Input details

The input should be in the form of an _x1dints product, which contains extracted spectra from multiple integrations for a given target.

Algorithm

The algorithm performs a simple sum of the flux values over all wavelengths for each extracted spectrum contained in the input product. If provided, min_wavelength and max_wavelength will modify the bounds of the sum to the specified bounds.

Output product

The output product is a table of time vs. integrated flux values, stored in the form of a ASCII ECSV (Extended Comma-Separated Value) file. The product type suffix is _whtlt.

Step Arguments

The white_light step has two step-specific arguments to allow wavelength limits during the flux summation. One or both may be specified.

--min_wavelength
: If min_wavelength is specified, the white_light step will sum from the specified wavelength to either a specified max_wavelength or the end of the flux array.

--max_wavelength
: If max_wavelength is specified, the white_light step will sum from either a specified min_wavelength or the beginning of the flux array to the specified wavelength.

Reference File

The white_light step uses the PARS-WHITELIGHTSTEP parameter reference file.

PARS-WHITELIGHTSTEP Parameter Reference File

REFTYPE
PARS-WHITELIGHTSTEP

Data model
N/A

Reference Selection Keywords

CRDS selects appropriate pars-whitelightstep references based on the following keywords.

Instrument	Keywords
MIRI	EXP_TYPE
NIRCAM	EXP_TYPE, FILTER, PUPIL
NIRSPEC	EXP_TYPE

Standard Keywords

The following table lists the keywords that are *required* to be present in all reference files. The first column gives the FITS keyword names. The second column gives the jwst data model name for each keyword, which is useful when using data models in creating and populating a new reference file. The third column gives the equivalent meta tag in ASDF reference file headers, which is the same as the name within the data model meta tree (second column).

FITS Keyword	Data Model Name	ASDF meta tag
AUTHOR	model.meta.author	author
DATAMODL	model.meta.model_type	model_type
DATE	model.meta.date	date
DESCRIP	model.meta.description	description
FILENAME	model.meta.filename	N/A
INSTRUME	model.meta.instrument.name	instrument: {name}
PEDIGREE	model.meta.pedigree	pedigree
REFTYPE	model.meta.reftype	reftype
TELESCOP	model.meta.telescope	telescope
USEAFTER	model.meta.useafter	useafter

NOTE: More information on standard required keywords can be found here: *Standard Required Keywords*

jwst.white_light Package

Classes

WhiteLightStep([name, parent, config_file, ...])	WhiteLightStep: Computes integrated flux as a function of time for a multi-integration spectroscopic observation.

WhiteLightStep

class jwst.white_light.**WhiteLightStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `JwstStep`

WhiteLightStep: Computes integrated flux as a function of time for a multi-integration spectroscopic observation.

Create a `Step` instance.

Parameters

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'white_light'

spec

```
min_wavelength     = float(default=None)      # Default wavelength minimum for␣
↪integration
max_wavelength     = float(default=None)      # Default wavelength maximum for␣
↪integration
output_ext         = string(default='.ecsv')  # Output file type
suffix             = string(default='whtlt')  # Default suffix for output files
```

Methods Documentation

process(*input*)

This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

Class Inheritance Diagram

PYTHON MODULE INDEX

j

INDEX

Symbols

A

B

C

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z