JWST Pipeline Documentation

Release 0.0.0.dev4172+gc5fd89fc

jwst

Oct 03, 2018

Contents

genindex | modindex

CHAPTER 1

Introduction

This document provides instructions on running the JWST Science Calibration Pipeline (referred to as "the pipeline") and individual pipeline steps.

Pipeline modules are available for detector-level (stage 1) processing of data from all observing modes, stage 2 processing for imaging and spectroscopic modes, and stage 3 processing for imaging, spectroscopic, coronagraphic, Aperture Masking Interferometry (AMI), and Time Series Observations (TSO).

Stage 1 processing consists of detector-level corrections that must be performed on a group-by-group basis before ramp fitting is applied. 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 *Stage 1 Pipeline Step Flow (calwebb_detector1)*.

Stage 2 processing consists of additional 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 *Stage 2 Imaging Pipeline Step Flow (calwebb_image2)* and *Stage 2 Spectroscopic Pipeline Step Flow (calwebb_spec2)*.

Stage 3 processing consists of routines that work with multiple exposures and in most cases produce some kind of combined product. There are dedicated (and unique) pipeline modules for stage 3 processing of imaging, spectroscopic, coronagraphic, AMI, and TSO observations. Details of each are available at *Stage 3 Imaging Pipeline Step Flow (calwebb_image3)*, *Stage 3 Spectroscopic Pipeline Step Flow (calwebb_spec3)*, *Stage 3 Coronagraphic Pipeline Step Flow (calwebb_coron3)*, *Stage 3 Aperture Masking Interferometry (AMI) Pipeline Step Flow (calwebb_ami3)*, and *Stage 3 Time-Series Observation(TSO) Pipeline Step Flow (calwebb_tso3)*.

The remainder of this document discusses pipeline configuration files and gives examples of running pipelines as a whole or in individual steps.

CHAPTER 2

Reference Files

Many pipeline steps rely on the use of a set of reference files essential to ensure the correct and accurate process of the data. The reference files are instrument-specific, and are periodically updated as the data process evolves and the understanding of the instruments improves. They are created, tested and validated by the JWST Instrument Teams. They 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 CRDS (the JWST Calibration Reference Data System), and made available to 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-formats-documentation as well as in the documentation for the Calibration Step using them.

CHAPTER 3

CRDS

CRDS reference file mappings are usually set by default to always give access to the most recent reference file deliveries and selection rules. 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'
```

The current storage location for all JWST CRDS reference files is:

```
/grp/crds/jwst/references/jwst/
```

Each pipeline step records the reference file that it used in the value of a header keyword in the output data file. The keyword names use the syntax "R_<ref>", where <ref> corresponds to a 6-character version of the reference file type, such as `R_DARK`, `R_LINEAR`, and `R_PHOTOM`.

CHAPTER 4

Running From the Command Line

Individual steps and pipelines (consisting of a series of steps) can be run from the command line using the `strun` command:

```
$ strun <class_name or cfg_file> <input_file>
```

The first argument to `strun` must be either the python class name of the step or pipeline to be run, or the name of a configuration (.cfg) file for the desired step or pipeline (see *Configuration Files* below for more details). The second argument to `strun` is the name of the input data file to be processed.

For example, running the full stage 1 pipeline or an individual step by referencing their class names is done as follows:

```
$ strun jwst.pipeline.Detector1Pipeline jw00017001001_01101_00001_nrca1_uncal.fits
$ strun jwst.dq_init.DQInitStep jw00017001001_01101_00001_nrca1_uncal.fits
```

When a pipeline or step is executed in this manner (i.e. by referencing the class name), it will be run using all default parameter values. The same thing can be accomplished by using the default configuration file corresponding to each:

```
$ strun calwebb_detector1.cfg jw00017001001_01101_00001_nrca1_uncal.fits
$ strun dq_init.cfg jw00017001001_01101_00001_nrca1_uncal.fits
```

If you want to use non-default parameter values, you can specify them as keyword arguments on the command line or set them in the appropriate cfg file. 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 jwst.pipeline.Detector1Pipeline jw00017001001_01101_00001_nrca1_uncal.fits
      --steps.dark_current.override_dark='my_dark.fits'
$ strun calwebb_detector1.cfg jw00017001001_01101_00001_nrca1_uncal.fits
      --steps.dark_current.override_dark='my_dark.fits'
```

You can get a list of all the available arguments for a given pipeline or step by using the '-h' (help) argument to strun:

```
$ strun dq_init.cfg -h
$ strun jwst.pipeline.Detector1Pipeline -h
```

If you want to consistently override the default values of certain arguments and don't want to specify them on the command line every time you execute `strun`, you can specify them in the configuration (.cfg) file for the pipeline or the individual step. For example, to always run `Detector1Pipeline` using the override in the previous example, you could edit your `calwebb_detector1.cfg` file to contain the following:

```
name = "Detector1Pipeline"
class = "jwst.pipeline.Detector1Pipeline"

   [steps]
     [[dark_current]]
       override_dark = 'my_dark.fits'
```

Note that simply removing the entry for a step from a pipeline cfg file will **NOT** cause that step to be skipped when you run the pipeline (it will simply run the step with all default parameters). In order to skip a step you must use the `skip = True` argument for that step (see *Skip* below).

Alternatively, you can specify arguments for individual steps within the step's configuration file and then reference those step cfg files in the pipeline cfg file, such as:

```
name = "Detector1Pipeline"
class = "jwst.pipeline.Detector1Pipeline"

   [steps]
     [[dark_current]]
       config_file = my_dark_current.cfg
```

where `my_dark_current.cfg` contains:

```
name = "dark_current"
class = "jwst.dark_current.DarkCurrentStep"
override_dark = 'my_dark.fits'
```

4.1 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 `calwebb_spec2` when, after successfully determining the WCS solution for a file, the WCS indicates that no science data will be found. This condition is most often found with NIRSpec's NRS2 detector. There are certain optical and MSA configurations in which dispersion will not cross to the NRS2 detector.

CHAPTER 5

Running From Within Python

You can execute a pipeline or a step from within python by using the `call` method of the class:

```
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')
```

The easiest way to use optional arguments when calling a pipeline from within python is to set those parameters in the pipeline cfg file and then supply the cfg file as a keyword argument:

```
Detector1Pipeline.call('jw00017001001_01101_00001_nrca1_uncal.fits', config_file=
↪'calwebb_detector1.cfg')
```

CHAPTER 6

Universal Parameters

6.1 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` argument. For example, to have all output from `calwebb_detector1`, including any saved intermediate steps, appear in the sub-directory `calibrated`, use

```
$ strun calwebb_detector1.cfg jw00017001001_01101_00001_nrca1_uncal.fits
    --output_dir=calibrated
```

`output_dir` can be specified at the step level, overriding what was specified for the pipeline. From the example above, to change the name and location of the `dark_current` step, use the following

```
$ strun calwebb_detector1.cfg jw00017001001_01101_00001_nrca1_uncal.fits
    --output_dir=calibrated
    --steps.dark_current.output_file='dark_sub.fits'
    --steps.dark_current.output_dir='dark_calibrated'
```

6.2 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:

- Specify `save_results`

  This option will save the results of the step, using a filename created by the step.

- Specify a file name using `output_file`

  This option will save the step results using the name specified.

For example, to save the result from the dark current step of `calwebb_detector1` in a file named `dark_sub.fits`, use

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

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

```
$ strun calwebb_detector1.cfg jw00017001001_01101_00001_nrca1_uncal.fits
    --output_file='detector1_processed.fits'
```

6.2.1 Output File and Associations

Stage 2 pipelines can take an individual file or an *association* as input. Nearly all Stage 3 pipelines require an associaiton as input. Normally, the output file is defined in each association's `product_name`.

If there is need to produce multiple versions of a calibration based on an association, 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. Stage 2 pipelines do not allow the override of the output using `output_file`. Stage 3 pipelines do. However, since Stage 3 pipelines generally produce many files per association, using different directories via `output_dir` will make file keeping simpler.

6.3 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. Arguments 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 arguments.

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

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

6.4 Skip

Another argument 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, edit the calwebb_detector1.cfg file to contain:

```
[steps]
   [[linearity]]
     skip = True
   ...
```

Alternatively you can specify the `skip` argument on the command line:

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

6.5 Logging Configuration

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. If you want log information saved to a file, you can specify the name of a logging configuration file either on the command line or in the pipeline cfg file.

For example:

```
$ strun calwebb_detector1.cfg 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.

CHAPTER 7

Input Files

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

The input data files - the exposure FITS files, association JSON files and input catalogs - are presumed to all be in the same directory as the primary input file. Sometimes the primary input is an association JSON file, and sometimes it is an exposure FITS file.

CHAPTER 8

Output File Names

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` argument

Regardless of the source, each pipeline/step uses the name as a "base name", on to which several different suffixes are appended, which indicate the type of data in that particular file.

8.1 Pipeline/Step Suffix Definitions

However the 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 existing 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.

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
1D extracted spectrum	x1d
1D extracted spectra per integration	x1dints
Resampled 2D image	i2d
Resampled 2D spectrum	s2d
Resampled 3D IFU cube	s3d
Source catalog	cat
Time Series photometric catalog	phot
Time Series white-light catalog	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 Individual Step Outputs

If individual steps are executed without an output file name specified via the `output_file` argument, 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 dq_init.cfg jw00017001001_01101_00001_nrca1_uncal.fits
```

produces an output file named `jw00017001001_01101_00001_nrca1_dq_init.fits`.

CHAPTER 9

Configuration Files

Configuration (.cfg) files can be used to specify parameter values when running a pipeline or individual steps, as well as for specifying logging options.

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.

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 "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 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
```

which specifies that all log messages will be directed to a file called "pipeline.log" and messages at a severity level of INFO and above will be recorded.

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. More information on configuration files can be found in the `stpipe` User's Guide at *For Users*.

CHAPTER 10

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 Modules*.

CHAPTER 11

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*.

CHAPTER 12

Package Documentation

12.1 Package Index

12.1.1 AMI Processing

Tasks in the Package

The Aperture Masking Interferometry (AMI) package currently consists of three tasks:

1) `ami_analyze`: apply the LG algorithm to a NIRISS AMI exposure
2) `ami_average`: average the results of LG processing for multiple exposures
3) `ami_normalize`: normalize the LG results for a science target using LG results from a reference target

The three tasks can be applied to an association of AMI exposures using the pipeline module `calwebb_ami3`.

CALWEBB_AMI3 Pipeline

Overview

The `calwebb_ami3` pipeline module can be used to apply all 3 steps of AMI processing to an association (ASN) of AMI exposures. The processing flow through the pipeline is as follows:

1) Apply the `ami_analyze` step to all products listed in the input association table. Output files will have a product type suffix of `ami`. There will be one `ami` product per input exposure.
2) Apply the `ami_average` step to combine the above results for both science target and reference target exposures, if both types exist in the ASN table. If the optional parameter `save_averages` is set to true (see below), the results will be saved to output files with a product type suffix of `amiavg`. There will be one `amiavg` product for the science target and one for the reference target.

3) If reference target results exist, apply the ami_normalize step to the averaged science target result, using the averaged reference target result to do the normalization. The output file will have a product type suffix of aminorm.

Input

The only input to the calwebb_ami3 pipeline is the name of a json-formatted association file. There is one optional parameter save_averages. If set to true, the results of the ami_average step will be saved to files. It is assumed that the ASN file will define a single output product for the science target result, containing a list of input member file names, for both science target and reference target exposures. An example ASN file is shown below.

```
{"asn_rule": "NIRISS_AMI", "targname": "NGC-3603", "asn_pool": "jw00017_001_01_pool",
↪"program": "00017",
"products": [
    {"prodtype": "ami", "name": "jw87003-c1001_t001_niriss_f277w-nrm",
     "members": [
        {"exptype": "science", "expname": "test_targ14_cal.fits"},
        {"exptype": "science", "expname": "test_targ15_cal.fits"},
        {"exptype": "science", "expname": "test_targ16_cal.fits"},
        {"exptype": "psf", "expname": "test_ref1_cal.fits"},
        {"exptype": "psf", "expname": "test_ref2_cal.fits"},
        {"exptype": "psf", "expname": "test_ref3_cal.fits"}]}],
"asn_type": "ami",
"asn_id": "c1001"}
```

Note that the exptype attribute value for each input member is used to indicate which files contain science target images and which contain reference psf images.

AMI_Analyze

Overview

The ami_analyze step 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 (on the fly) a region from the image corresponding to the size and location of the SUB80 subarray, in order to keep the processing time to a reasonable level.

Inputs

The ami_analyze step takes a single input image, in the form of a simple 2D ImageModel. There are two optional parameters:

1) oversample: specifies the oversampling factor to be used in the model fit (default value = 3)

2) rotation: specifies an initial guess, in degrees, for the rotation of the PSF in the input image (default value = 0.0)

Output

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

1) FIT: a 2-D image of the fitted model
2) RESID: a 2-D 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

AMI_Average

Overview

The ami_average step averages the results of LG processing from the ami_analyze step for multiple exposures of a given target. It averages all 8 components of the ami_analyze output files for all input exposures.

Inputs

The only input to the ami_average step is a list of input files to be processed. These will presumably be output files from the ami_analyze step. The step has no other required or optional parameters, nor does it use any reference files.

Output

The step produces a single output file, having the same format as the input files, where the data for the 8 file components are the average of each component from the list of input files.

AMI_Normalize

Overview

The ami_normalize step provides normalization of LG processing results for a science target using LG results of a reference target. The algorithm subtracts the reference target closure phases from the science target closure phases and divides the science target fringe amplitudes by the reference target fringe amplitudes.

Inputs

The ami_normalize step takes two input files: the first is the LG processed results for a science target and the second is the LG processed results for the reference target. There are no optional parameters and no reference files are used.

Output

The output is a new LG product for the science target in which the closure phases and fringe amplitudes have been normalized using the reference target closure phases and fringe amplitudes. The remaining components of the science target data model are left unchanged.

Reference File Types

The `ami_analyze` step uses a THROUGHPUT reference file, which contains throughput data for the filter used in the input AMI image. (The ami_average and ami_normalize steps do not use any reference files.)

CRDS Selection Criteria

Throughput reference files are selected on the basis of INSTRUME and FILTER values for the input science data set.

Throughput Reference File Format

Throughput reference files are FITS files with one BINTABLE extension. The FITS primary data array is assumed to be empty. The table extension uses `EXTNAME=THROUGHPUT` and the data table has the following characteristics:

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

jwst.ami Package

Classes

AmiAnalyzeStep([name, parent, config_file, ...])	AmiAnalyzeStep: Performs analysis of an AMI mode exposure by applying the LG algorithm.
AmiAverageStep([name, parent, config_file, ...])	AmiAverageStep: Averages LG results for multiple NIRISS AMI mode exposures
AmiNormalizeStep([name, parent, ...])	AmiNormalizeStep: Normalize target LG results using reference LG results

AmiAnalyzeStep

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

Bases: *jwst.stpipe.Step*

AmiAnalyzeStep: 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.
- ****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

reference_file_types
spec

Methods Summary

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

Attributes Documentation

reference_file_types = ['throughput']

spec = '\n oversample = integer(default=3, min=1) # Oversampling factor\n rotation = f

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

AmiAverageStep

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

Bases: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

process(input_list)	Averages the results of LG analysis for a set of multiple NIRISS AMI mode exposures.

Attributes Documentation

spec = '\n '

Methods Documentation

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

AmiNormalizeStep

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

Bases: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

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

Attributes Documentation

spec = '\n '

Methods Documentation

process(*target*, *reference*)

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

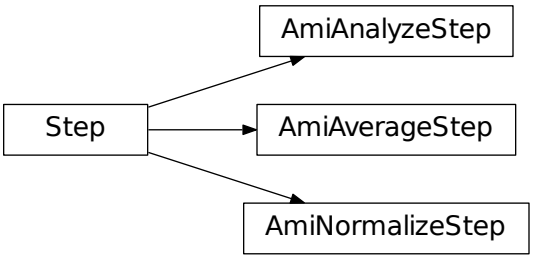
Parameters

- **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



12.1.2 Assign WCS

Description

`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`.

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.

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 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:

```
>>> form jwst.datamodels import ImageModel
>>> exp = ImageModel('nircam_wfss_assign_wcs.fits')
>>> x, y, x0, y0, order = exp.meta.wcs(x0, y0, wavelength, order)
```

(continues on next page)

(continued from previous page)

```
>>> 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 = models.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 any type of DataModel as input. In particular, for multiple-integration datasets the step will accept either of these data products: the slope results for each integration in the exposure, or the single slope image that is the result of averaging over all integrations.

`jwst.assign_wcs` is based on gwcs and uses the modeling, units and coordinates subpackages in astropy.

Software dependencies:

- gwcs (https://github.com/spacetelescope/gwcs) 0.7
- numpy (http://www.numpy.org/) 1.9 or later
- astropy (http://www.astropy.org/) 1.2.1 or later
- asdf (http://asdf.readthedocs.io/en/latest/) 1.1.1 or later

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.

List of reference types used by assign_wcs

reftype	description	Instrument
camera	NIRSPEC Camera model	NIRSPEC
collimator	NIRSPEC Collimator Model	NIRSPEC
disperser	Disperser parameters	NIRSPEC
distortion	Spatial distortion model	MIRI, FGS, NIRCAM, NIRISS
filteroffset	MIRI Imager fiter offsets	MIRI
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	FU Slicer geometric description	NIRSPEC
msa	Transformin 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, NIRSPEC, NIRCAM, NIRISS

CRDS Selection Criteria

CAMERA (NIRSPEC only)

CAMERA reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

COLLIMATOR (NIRSPEC only)

For NIRSPEC, COLLIMATOR reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

DISPERSER (NIRSPEC only)

For NIRSPEC, DISPERSER reference files are currently selected based on the values of EXP_TYPE and GRATING in the input science data set.

DISTORTION

For MIRI, DISTORTION reference files are currently selected based on the values of EXP_TYPE, DETECTOR, CHANNEL, and BAND in the input science data set.

For FGS, DISTORTION reference files are currently selected based on the values of EXP_TYPE and DETECTOR in the input science data set.

For NIRCAM, DISTORTION reference files are currently selected based on the values of EXP_TYPE, DETECTOR, CHANNEL, and FILTER in the input science data set.

For NIRISS, DISTORTION reference files are currently selected based only on the value of EXP_TYPE and PUPIL in the input science data set.

FILTEROFFSET (MIRI only)

For MIRI, FILTEROFFSET reference files are currently selected based on the values of EXP_TYPE and DETECTOR in the input science data set.

FORE (NIRSPEC only)

For NIRSPEC, FORE reference files are currently selected based on the values of EXP_TYPE and FILTER in the input science data set.

FPA (NIRSPEC only)

For NIRSPEC, FPA reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

IFUFORE (NIRSPEC only)

For NIRSPEC, IFUFORE reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

IFUPOST (NIRSPEC only)

For NIRSPEC, IFUPOST reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

IFUSLICER (NIRSPEC only)

For NIRSPEC, IFUSLICER reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

MSA (NIRSPEC only)

For NIRSPEC, MSA reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

OTE (NIRSPEC only)

For NIRSPEC, OTE reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

SPECWCS

For MIRI, SPECWCS reference files are currently selected based on the values of DETECTOR, CHANNEL, BAND, SUBARRAY, and EXP_TYPE in the input science data set.

For NIRCAM, SPECWCS reference files are currently selected based on the values of EXP_TYPE, MODULE, and PUPIL in the input science data set.

For NIRCAM WFSS, SPECWCS reference files are currently selected based on the values of EXP_TYPE, MODULE, and PUPIL in the input science data set.

For NIRCAM TGRISM, SPECWCS reference files are currently selected based on the values of EXP_TYPE, MODULE, and PUPIL in the input science data set.

FOR NIRISS WFSS, SPECWCS reference files are currently selected based on the values of EXP_TYPE, FILTER, and PUPIL in the input science data set.

REGIONS (MIRI only)

For MIRI, REGIONS reference files are currently selected based on the values of DETECTOR, CHANNEL, BAND, and EXP_TYPE in the input science data set.

WAVELENGTHRANGE

For NIRCAM, NIRISS, NIRSPEC, and MIRI, WAVELENGTHRANGE reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

Reference File Formats

CAMERA

The camera reference file contains an astropy compound model made up of polynomial models, rotations, and translations. The forward direction is from the FPA to the GWA.

model Transform through the CAMERA.

COLLIMATOR

This collimator reference file contains an astropy compound model made up of polynomial models, rotations, and translations. The forward direction is from the GWA to the MSA.

model Transform through the COLLIMATOR.

DISPERSER

The disperser reference file contains reference data about the NIRSPEC dispersers (gratings or the prism).

Files applicable to gratings have a field:

groovedensity Number of grooves per meter in a grating

The following fields are common for all gratings and the prism:

grating Name of grating

gwa_tiltx

temperatures Temperatures measured where the GWA sensor is

zeroreadings Value of GWA sensor reading which corresponds to disperser model parameters

tilt_model Model of the relation between THETA_Y vs GWA_X sensor reading

gwa_tilty

temperatures Temperatures measured where the GWA sensor is

zeroreadings Value of GWA sensor reading which corresponds to disperser model parameters

tilt_model Model of the relation between THETA_X vs GWA_Y sensor reading

tilt_x Angle (in degrees) between the grating surface and the reference surface (the mirror)

tilt_y Angle (in degrees) between the grating surface and the reference surface (the mirror)

theta_x Element alignment angle in x-axis (in degrees)

theta_y Element alignment angle in y-axis (in degrees)

theta_z Element alignment angle in z-axis (in degrees)

The prism reference file has in addition the following fields:

angle Angle between the front and back surface of the prosm (in degrees)

kcoef K coefficients of Selmeir equation, describing the material

lcoef L coeffficients describing the material

tcoef Six constants, describing the thermal behavior of the glass

tref Temperature (in K), used to compute the change in temperature relative to the reference temperature of the glass

pref Reference pressure (in ATM)

wbound Min and Max wavelength (in meters) for which the model is valid

DISTORTION

The distortion reference file contains a combination of astropy models, representing the transform from detector to the telescope V2, V3 system. The following convention was adopted:

- The output in the V2, V3 system is in units of arcsec.
- The input x and y are 0-based coordinates in the DMS system.
- The center of the first pixel is (0, 0), so the first pixel goes from -0.5 to 0.5.
- The origin of the transform is taken to be (0, 0). Note, that while a different origin can be used for some transforms the relevant offset should first be prepended to the distortion transform to account for the change in origin of the coordinate frame. For instance, MIRI takes input in (0, 0) - indexed detector pixel coordinates, but shifts these around prior to calling transforms that are defined with respect to science-frame pixels that omit reference pixels.

Internally the WCS pipeline works with 0-based coordinates. When FITS header keywords are used, the 1 pixel offset in FITS coordinates is accounterd for internally in the pipeline.

The model is a combination of polynomials.

model Transform from detector to an intermediate frame (instrument dependent).

FILTEROFFSET

The filter offset reference file is an ASDF file that contains a dictionary of row and column offsets for the MIRI imaging dataset. The filter offset reference file contains a dictionary in the tree that is indexed by the instrument filter. Each filter points to two fields - row_offset and column_offset. The format is

miri_filter_name

column_offset Offset in x (in arcmin)

row_offset Offset in y (in arcmin)

FORE

The FORE reference file stores the transform through the Filter Wheel Assembly (FWA). It has two fields - "filter" and "model". The transform through the FWA is chromatic. It is represented as a Polynomial of two variables whose coefficients are wavelength dependent. The compound model takes three inputs - x, y positions and wavelength.

filter Filter name.

model Transform through the Filter Wheel Assembly (FWA).

FPA

The FPA reference file stores information on the metrology of the Focal Plane Assembly (FPA) which consists of two Sensor Chip Arrays (SCA), named NRS1 and NRS2.

The reference file contains two fields : "nrs1_model" and "nrs2_model". Each of them stores the transform (shift and rotation) to transform positions from the FPA to the respective SCA. The output units are in pixels.

nrs1_model Transform for the NRS1 detector.

nrs2_model Transform for the NRS2 detector.

IFUFORE

This file provides the parameters (Paraxial and distortions coefficients) for the coordinate transforms from the MSA plane to the plane of the IFU slicer.

model Compound model, Polynomials

IFUPOST

The IFUPOST reference file provides the parameters (Paraxial and distortions coefficients) for the coordinate transforms from the slicer plane to the MSA plane (out), that is the plane of the IFU virtual slits.

The reference file contains models made up based on an offset and a polynomial. There is a model for each of the slits and is indexed by the slit number. The models is used as part of the conversion from the GWA to slit.

slice_<slice_number>

model Polynomial and rotation models.

IFUSLICER

The IFUSLICER stores information about the metrology of the IFU slicer - relative positioning and size of the aperture of each individual slicer and the absolute reference with respect to the center of the field of view. The reference file contains two fields - "data" and "model". The "data" field is an array with 30 rows pertaining to the 30 slices and the columns are

data Array with reference data for each slicer. It has 5 columns

NO Slice number (0 - 29)

x_center X coordinate of the center (in meters)

y_center Y coordinate of the center (in meters)

x_size X size of teh aperture (in meters)

y_size Y size of the aperture (in meters)

model Transform from relative positions within the IFU slicer to absolute positions within the field of view. It's a combination of shifts and rotation.

MSA

The MSA reference file contains information on the metrology of the microshutter array and the associated fixed slits - relative positioning of each individual shutter (assumed to be rectangular) And the absolute position of each quadrant within the MSA.

The MSA reference file has 5 fields, named

1

data Array with reference data for each shutter in Quadrant 1. It has 5 columns

NO Shutter number (1- 62415)

x_center X coordinate of the center (in meters)

y_center Y coordinate of the center (in meters)

x_size X size of teh aperture (in meters)

y_size Y size of the aperture (in meters)

model Transform from relative positions within Quadrant 1 to absolute positions within the MSA

2

data Array with reference data for shutters in Quadrant 2, same as in 1 above

model Transform from relative positions within Quadrant 2 to absolute positions within the MSA

3

data Array with reference data for shutters in Quadrant 3, same as in 1 above

model Transform from relative positions within Quadrant 3 to absolute positions within the MSA

4

data Array with reference data for shutters in Quadrant 4, same as in 1 above

model Transform from relative positions within Quadrant 4 to absolute positions within the MSA

5

data Reference data for the fixed slits and the IFU, same as in 1, except NO is 6 rows (1-6) and the mapping is 1 - S200A1, 2 - S200A1, 3 - S400A1, 4 - S200B1, 5 - S1600A1, 6 - IFU

model Transform from relative positions within eac aperture to absolute positions within the MSA

OTE

This reference file contains a combination of astropy models - polynomial, shift, rotation and scaling.

model Transform through the Optical Telescope Element (OTE), from the FWA to XAN, YAN telescope frame. The output units are in arcsec.

SPECWCS

For the MIRI LRS mode the file is in FITS format. The reference file contains the zero point offset for the slit relative to the full field of view. For the Fixed Slit exposure type the zero points in X and Y are stored in the header of the second HDU in the 'IMX' and 'IMY' keywords. For the Slitless exposure type they are stored in the header of the second HDU in FITS keywords 'IMXSLTl' and 'IMYSLTl'. For both of the exposure types, the zero point offset is 1 based and the X (e.g., IMX) refers to the column and Y refers to the row.

For the MIRI MRS the file is in ASDF format with the following structure.

channel The MIRI channels in the observation, e.g. "12".

band The band for the observation (one of "LONG", "MEDIUM", "SHORT").

model

slice_number The wavelength solution for each slice. <slice_number> is the actual slice number (s), computed by s = channel * 100 + slice

For NIRISS SOSS mode the file is in ASDF format with the following structure.

model A tabular model with the wavelength solution.

For NIRCAM WFSS and TSGRISM modes the file is in ASDF format with the following structure:

displ The wavelength transform models

dispx The x-dispersion models

dispy The y-dispersion models

invdispx The inverse x-dispersion models

invdispy The inverse y-dispersion models

invdispl The inverse wavelength transform models

orders a list of order numbers that the models relate to, in the same order as the models

For NIRISS WFSS mode the file is in ASDF format with the following structure:

displ The wavelength transform models

dispx The x-dispersion models

dispy The y-dispersion models

invdispx The inverse x-dispersion models

invdispl The inverse wavelength transform models

fwcpos_ref The reference filter wheel position in degrees

orders a list of order numbers that the models relate to, in the same order as the models

Regions

The IFU takes a region reference file that defines the region over which the WCS is valid. The reference file should define a polygon and may consist of a set of X,Y coordinates that define the polygon.

channel The MIRI channels in the observation, e.g. "12".

band The band for the observation (one of "LONG", "MEDIUM", "SHORT").

regions An array with the size of the MIRI MRS image where pixel values map to the MRS slice number. 0 indicates a pixel is not within any slice.

WAVELENGTHRANGE

FOR MIRI MRS the wavelengthrange file consists of two fields which define te wavelength range for each combination of a 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

For NIRSPEC the 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

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 | scale_x
>>> 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, 29242.892)
```

`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, 1)
    -152.2
```

Create the reference file

The DictortionModel 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 validaiton of the file structure.

```
>>> from jwst.datamodels import DistortionModel
>>> dist = DistortionModel(model=model)
>>> dist.validate()
>>> dist.save("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 is create using the rules in the above section, it needs to be assigned to the ASDF tree.

```
>>> 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 `AsdfFile.open()` method:

```
>>> ff = AsdfFile.open('reffile.asdf')
>>> model = ff.tree['model']
>>> model(1, 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*
WCS pipeline coordinate frames: detector, v2v3, world
Implements: Distortion file created from TEL team data.

NRC_WFSS, NRC_TSGRISM

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

NIS_WFSS

reftypes: *specwcs*, *distortion*
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

12.1.3 Reference/API

jwst.assign_wcs.fgs Module

FGS WCS pipeline - depends on EXP_TYPE.

Functions

create_pipeline(input_model, reference_files)	Create a `gWCS.pipeline` using models from reference files.
imaging(input_model, reference_files)	The FGS imaging WCS pipeline.

create_pipeline

jwst.assign_wcs.fgs.**create_pipeline**(*input_model*, *reference_files*)
Create a `gWCS.pipeline` using models from reference files.

Parameters

- **input_model** (`jwst.datamodels.DataModel`) – Either an ImageModel or a CubeModel
- **reference_files** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – {reftype: file_name} mapping. Reference files.

imaging

jwst.assign_wcs.fgs.**imaging**(*input_model*, *reference_files*)
The FGS imaging WCS pipeline.

It includes 3 coordinate frames - "detector", "v2v3" and "world".

Uses a `distortion` reference file.

jwst.assign_wcs.miri Module

Functions

create_pipeline(input_model, reference_files)	Create the WCS pipeline for MIRI modes.
imaging(input_model, reference_files)	The MIRI Imaging WCS pipeline.
lrs(input_model, reference_files)	The LRS-FIXEDSLIT and LRS-SLITLESS WCS pipeline.
ifu(input_model, reference_files)	The MIRI MRS WCS pipeline.

create_pipeline

jwst.assign_wcs.miri.**create_pipeline**(*input_model*, *reference_files*)

Create the WCS pipeline for MIRI modes.

Parameters

- **input_model** (jwst.datamodels.ImagingModel, *IFUImageModel*, *CubeModel*) – Data model.
- **reference_files** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – {reftype: reference file name} mapping.

imaging

jwst.assign_wcs.miri.**imaging**(*input_model*, *reference_files*)

The MIRI Imaging WCS pipeline.

It includes three coordinate frames - “detector”, “v2v3” and “world”.

Parameters

- **input_model** (jwst.datamodels.ImagingModel) – Data model.
- **reference_files** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Dictionary {reftype: reference file name}. Uses “distortion” and “filteroffset” reference files.

lrs

jwst.assign_wcs.miri.**lrs**(*input_model*, *reference_files*)

The LRS-FIXEDSLIT and LRS-SLITLESS WCS pipeline.

It has two coordinate frames: “detecor” and “world”. Uses the “specwcs” and “distortion” reference files.

ifu

jwst.assign_wcs.miri.**ifu**(*input_model*, *reference_files*)

The MIRI MRS WCS pipeline.

It has the following coordinate frames: “detector”, “alpha_beta”, “v2v3”, “world”.

It uses the “distortion”, “regions”, “specwcs” and “wavelengthrange” reference files.

jwst.assign_wcs.nircam Module

Functions

create_pipeline(input_model, reference_files)	Create the WCS pipeline based on EXP_TYPE.
imaging(input_model, reference_files)	The NIRCAM imaging WCS pipeline.
tsgrism(input_model, reference_files)	Create WCS pipeline for a NIRCAM Time Series Grism observation.
wfss(input_model, reference_files)	Create the WCS pipeline for a NIRCAM grism observation.

create_pipeline

jwst.assign_wcs.nircam.**create_pipeline**(*input_model*, *reference_files*)
Create the WCS pipeline based on EXP_TYPE.

imaging

jwst.assign_wcs.nircam.**imaging**(*input_model*, *reference_files*)
The NIRCAM imaging WCS pipeline.

It includes three coordinate frames - “detector”, “v2v3” and “world”.

It uses the “distortion” reference file.

tsgrism

jwst.assign_wcs.nircam.**tsgrism**(*input_model*, *reference_files*)
Create WCS pipeline for a NIRCAM Time Series Grism observation.

Parameters

- **input_model** (*jwst.datamodels.ImagingModel*) – The input datamodel, derived from datamodels
- **reference_files** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Dictionary {reftype: reference file name}.

Notes

The TSGRISM mode should function effectively like the grism mode except that subarrays will be allowed. Since the transform models depend on the original full frame coordinates of the observation, the regular grism transforms will need to be shifted to the full frame coordinates around the trace transform.

TSGRISM is only slated to work with GRISMR and Mod A

wfss

jwst.assign_wcs.nircam.**wfss**(*input_model*, *reference_files*)
Create the WCS pipeline for a NIRCAM grism observation.

Parameters

- **input_model** (*jwst.datamodels.ImagingModel*) – The input datamodel, derived from datamodels
- **reference_files** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Dictionary {reftype: reference file name}.

Notes

The tree in the grism reference file has a section for each order/beam not sure if there will be a separate passband reference file needed for the wavelength scaling or wedge offsets. This helper is currently in jwreftools/nircam/nircam_reftools.

The direct image the catalog has been created from was corrected for distortion, but the dispersed images have not. This is OK if the trace and dispersion solutions are defined with respect to the distortion-corrected image. The catalog from the combined direct image has object locations in in detector space and the RA DEC of the object on sky.

The WCS information for the grism image plus the observed filter will be used to translate these to pixel locations for each of the objects. The grism images will then use their grism trace information to translate to detector space. The translation is assumed to be one-to-one for purposes of identifying the center of the object trace.

The extent of the trace for each object can then be calculated based on the grism in use (row or column). Where the left/bottom of the trace starts at t = 0 and the right/top of the trace ends at t = 1, as long as they have been defined as such by th team.

The extraction box is calculated to be the minimum bounding box of the object extent in the segmentation map associated with the direct image. The values of the min and max corners are saved in the photometry catalog in units of RA,DEC so they can be translated to pixels by the dispersed image's imaging wcs.

For each spectral order, the configuration file contains a magnitude-cutoff value. Sources with magnitudes fainter than the extraction cutoff (MMAG_EXTRACT) will not be extracted, but are accounted for when computing the spectral contamination and background estimates. The default extraction value is 99 right now.

The sensitivity information from the original aXe style configuration file needs to be modified by the passband of the filter used for the direct image to get the min and max wavelengths which correspond to t=0 and t=1, this currently has been done by the team and the min and max wavelengths to use to calculate t are stored in the grism reference file as wavelengthrange, which can be selected by waverange_selector which contains the filter names.

All the following was moved to the extract_2d stage.

Step 1: Convert the source catalog from the reference frame of the uberimage to that of the dispersed image. For the Vanilla Pipeline we assume that the pointing information in the file headers is sufficient. This will be strictly true if all images were obtained in a single visit (same guide stars).

Step 2: Record source information for each object in the catalog: position (RA and Dec), shape (A_IMAGE, B_IMAGE, THETA_IMAGE), and all available magnitudes.

Step 3: Compute the trace and wavelength solutions for each object in the catalog and for each spectral order. Record this information.

Step 4: Compute the WIDTH of each spectral subwindow, which may be fixed or variable (see discussion of optimal extraction, below). Record this information.

Catalog and associated steps moved to extract_2d.

jwst.assign_wcs.niriss Module

Functions

create_pipeline(input_model, reference_files)	Create the WCS pipeline based on EXP_TYPE.
imaging(input_model, reference_files)	The NIRISS imaging WCS pipeline.
niriss_soss(input_model, reference_files)	The NIRISS SOSS WCS pipeline.
niriss_soss_set_input(model, order_number)	Extract a WCS fr a specific spectral order.
wfss(input_model, reference_files)	Create the WCS pipeline for a NIRISS grism observation.

create_pipeline

jwst.assign_wcs.niriss.**create_pipeline**(*input_model*, *reference_files*)
Create the WCS pipeline based on EXP_TYPE.

imaging

jwst.assign_wcs.niriss.**imaging**(*input_model*, *reference_files*)
The NIRISS imaging WCS pipeline.

It includes three coordinate frames - “detector” “v2v3” and “world”.

It uses the “distortion” reference file.

niriss_soss

jwst.assign_wcs.niriss.**niriss_soss**(*input_model*, *reference_files*)
The NIRISS SOSS WCS pipeline.

It includes tWO coordinate frames - “detector” and “world”.

It uses the “specwcs” reference file.

niriss_soss_set_input

jwst.assign_wcs.niriss.**niriss_soss_set_input**(*model*, *order_number*)
Extract a WCS fr a specific spectral order.

Parameters

- **– ImageModel** (*model*) –
- **– the spectral order** (*order_number*) –

Returns

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

wfss

jwst.assign_wcs.niriss.**wfss**(*input_model*, *reference_files*)
Create the WCS pipeline for a NIRISS grism observation.

Parameters

- **input_model** (*jwst.datamodels.ImagingModel*) – The input datamodel, derived from datamodels
- **reference_files** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Dictionary specifying reference file names

Notes

reference_files = { “specwcs”: ‘GR150C_F090W.asdf’ “distortion”: ‘NRCA1_FULL_distortion.asdf’ }

The tree in the grism reference file has a section for each order/beam as well as the link to the filter data file, not sure if there will be a separate passband reference file needed for the wavelength scaling or the wedge offsets. This file is currently created in jwreftools/niriss/niriss_reftools.

The direct image the catalog has been created from was corrected for distortion, but the dispersed images have not. This is OK if the trace and dispersion solutions are defined with respect to the distortion-corrected image. The catalog from the combined direct image has object locations in in detector space and the RA DEC of the object on sky.

The WCS information for the grism image plus the observed filter will be used to translate these to pixel locations for each of the objects. The grism images will then use their grism trace information to translate to detector space. The translation is assumed to be one-to-one for purposes of identifying the center of the object trace.

The extent of the trace for each object can then be calculated based on the grism in use (row or column). Where the left/bottom of the trace starts at t = 0 and the right/top of the trace ends at t = 1, as long as they have been defined as such by th team.

The extraction box is calculated to be the minimum bounding box of the object extent in the segmentation map associated with the direct image. The values of the min and max corners are saved in the photometry catalog in units of RA,DEC so they can be translated to pixels by the dispersed image’s imaging wcs.

The sensitivity information from the original aXe style configuration file needs to be modified by the passband of the filter used for the direct image to get the min and max wavelengths which correspond to t=0 and t=1, this currently has been done by the team and the min and max wavelengths to use to calculate t are stored in the grism reference file as wrange, which can be selected by wrange_selector which contains the filter names.

Source catalog use moved to extract_2d.

jwst.assign_wcs.nirspec Module

Tools to create the WCS pipeline NIRSPEC modes.

Calls create_pipeline() which redirects based on EXP_TYPE.

Functions

create_pipeline(input_model, reference_files)	Create a pipeline list based on EXP_TYPE.
imaging(input_model, reference_files)	Imaging pipeline.
ifu(input_model, reference_files)	The Nirspec IFU WCS pipeline.
slits_wcs(input_model, reference_files)	The WCS pipeline for MOS and fixed slits.
get_open_slits(input_model[, reference_files])	Return the opened slits/shutters in a MOS or Fixed Slits exposure.

Continued on next page

Table 12 – continued from previous page

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.

create_pipeline

jwst.assign_wcs.nirspec.**create_pipeline**(*input_model*, *reference_files*)
Create a pipeline list based on EXP_TYPE.

Parameters

- **input_model** (*ImageModel*, *IFUImageModel*, *CubeModel*) – The input exposure.
- **reference_files** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – {reftype: reference_file_name} mapping.

imaging

jwst.assign_wcs.nirspec.**imaging**(*input_model*, *reference_files*)
Imaging pipeline.

It has the following coordinate frames: "detector" : the science frame "sca" : frame associated with the SCA "gwa" " just before the GWA going from detector to sky "msa_frame" : at the MSA "oteip" : after the FWA "v2v3" and "world"

ifu

jwst.assign_wcs.nirspec.**ifu**(*input_model*, *reference_files*)
The Nirspec IFU WCS pipeline.

The coordinate frames are: "detector" : the science frame "sca" : frame associated with the SCA "gwa" " just before the GWA going from detector to sky "slit_frame" : frame associated with the virtual slit "slicer' : frame associated with the slicer "msa_frame" : at the MSA "oteip" : after the FWA "v2v3" and "world"

slits_wcs

jwst.assign_wcs.nirspec.**slits_wcs**(*input_model*, *reference_files*)
The WCS pipeline for MOS and fixed slits.

The coordinate frames are: "detector" : the science frame "sca" : frame associated with the SCA "gwa" " just before the GWA going from detector to sky "slit_frame" : frame associated with the virtual slit "msa_frame" : at the MSA "oteip" : after the FWA "v2v3" : at V2V3 "world" : sky and spectral

get_open_slits

jwst.assign_wcs.nirspec.**get_open_slits**(*input_model*, *reference_files=None*)
Return the opened slits/shutters in a MOS or Fixed Slits exposure.

nrs_wcs_set_input

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

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

Parameters

- **input_model** (*DataModel*) – 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 fliter and grating.

Returns **wcsobj** – WCS object for this slit.

Return type WCS

nrs_ifu_wcs

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

Return a list of WCSs for all NIRSPEC IFU slits.

Parameters **input_model** (jwst.datamodels.DataModel) – The data model. Must have been through the assign_wcs step.

get_spectral_order_wrange

jwst.assign_wcs.nirspec.**get_spectral_order_wrange**(*input_model*, *wavelength_range_file*)

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

Parameters

- **input_model** (*DataModel*) – The input data model.
- **wavelengthrange_file** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Reference file of type "wavelengthrange".

jwst.assign_wcs.pointing Module

Functions

compute_roll_ref(v2_ref, v3_ref, roll_ref, ...)	Computes the position of V3 (measured N to E) at the center af an aperture.
frame_from_model(wcsinfo)	Initialize a coordinate frame based on values in model.meta.wcsinfo.
fitswcs_transform_from_model(wcsinfo[, wavetab])	Create a WCS object using from datamodel.meta.wcsinfo.

compute_roll_ref

jwst.assign_wcs.pointing.**compute_roll_ref**(*v2_ref*, *v3_ref*, *roll_ref*, *ra_ref*, *dec_ref*, *new_v2_ref*, *new_v3_ref*)

Computes the position of V3 (measured N to E) at the center af an aperture.

Parameters

- **v3_ref** (*v2_ref,*) – Reference point in the V2, V3 frame [in arcsec] (FITS keywords V2_REF and V3_REF)
- **roll_ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – Position angle of V3 at V2_REF, V3_REF, [in deg] When ROLL_REF == PA_V3, then (V2_REF, V3_REF) = (0, 0)
- **dec_ref** (*ra_ref,*) – RA and DEC corresponding to V2_REF and V3_REF, [in deg]
- **new_v3_ref** (*new_v2_ref,*) – The new position in V2, V3 where the position of V3 is computed, [in arcsec] The center of the aperture in V2,V3

Returns **new_roll** – The value of ROLL_REF (in deg)

Return type float (https://docs.python.org/3/library/functions.html#float)

frame_from_model

jwst.assign_wcs.pointing.**frame_from_model**(*wcsinfo*)

Initialize a coordinate frame based on values in model.meta.wcsinfo.

Parameters **wcsinfo** (*DataModel* or dict) – Either one of the JWST data moels or a dict with model.meta.wcsinfo.

Returns **frame**

Return type CoordinateFrame

fitswcs_transform_from_model

jwst.assign_wcs.pointing.**fitswcs_transform_from_model**(*wcsinfo*, *wavetab=None*)

Create a WCS object using from datamodel.meta.wcsinfo. Transforms assume 0-based coordinates.

Parameters **wcsinfo** (*dict-like*) – ~jwst.meta.wcsinfo structure.

Returns **transform** – WCS forward transform - from pixel to world coordinates.

Return type Model

jwst.assign_wcs.util Module

Utility function for assign_wcs.

Functions

reproject(wcs1, wcs2[, origin])	Given two WCSs return a function which takes pixel coordinates in the first WCS and computes their location in the second one.
wcs_from_footprints(dmodels[, refmodel, ...])	Create a WCS from a list of input data models.
velocity_correction(velosys)	Compute wavelength correction to Barycentric reference frame.

reproject

jwst.assign_wcs.util.**reproject**(*wcs1*, *wcs2*, *origin=0*)

Given two WCSs return a function which takes pixel coordinates in the first WCS and computes their location in the second one.

It performs the forward transformation of `wcs1` followed by the inverse of `wcs2`.

Parameters **wcs2** (*wcs1,*) – WCS objects.

Returns **_reproject** – Function to compute the transformations. It takes x, y positions in `wcs1` and returns x, y positions in `wcs2`.

Return type func

wcs_from_footprints

jwst.assign_wcs.util.**wcs_from_footprints**(*dmodels*, *refmodel=None*, *transform=None*, *bounding_box=None*, *domain=None*)

Create a WCS from a list of input data models.

A fiducial point in the output coordinate frame is created from the footprints of all WCS objects. For a spatial frame this is the center of the union of the footprints. For a spectral frame the fiducial is in the beginning of the footprint range. If `refmodel` is None, the first WCS object in the list is considered a reference. The output coordinate frame and projection (for celestial frames) is taken from `refmodel`. If `transform` is not suplied, a compound transform is created using CDELTs and PC. If `bounding_box` is not supplied, the bounding_box of the new WCS is computed from bounding_box of all input WCSs.

Parameters

- **dmodels** (list of *DataModel*) – A list of data models.
- **refmodel** (*DataModel*, optional) – This model's WCS is used as a reference. WCS. The output coordinate frame, the projection and a scaling and rotation transform is created from it. If not supplied the first model in the list is used as `refmodel`.
- **transform** (`Model`, optional) – A transform, passed to `wcs_from_fiducial()` If not supplied Scaling | Rotation is computed from `refmodel`.
- **bounding_box** (*tuple* (https://docs.python.org/3/library/stdtypes.html#tuple), *optional*) – Bounding_box of the new WCS. If not supplied it is computed from the bounding_box of all inputs.

velocity_correction

jwst.assign_wcs.util.**velocity_correction**(*velosys*)

Compute wavelength correction to Barycentric reference frame.

Parameters **velosys** (*float* (https://docs.python.org/3/library/functions.html#float)) – Radial velocity wrt Barycenter [m / s].

12.1.4 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 occured 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 two stages: Level2 processing and Level3 processing. Level2 processing is the calibration necessary to remove instrumental effects from the data. The resulting files contain flux and spatially calibrated data, called `Level2b` data. The information is still in individual exposures.

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 `Level3` processing. Due to the nature of the individual instruments, observing modes, and the interruptability 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 Level3 calibration steps to perform the transformation from exposure-based data to source-based, high(er) signal-to-noise data.

In short, Level 2 and Level 3 associations are created running the *asn_generate* task on an Association Pool using the default Level 2 and *Level 3 Association Rules* to produce level2-associations and level3-associations.

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. Or, if need be, an association can be created based on the existing *Level2* or *Level3* examples.

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

```
% strun calwebb_image3.cfg  jw12345_xxxx_asn.json
```

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

```
from jwst.associations import load_asn

with open('jw12345_xxxx_asn.json') as fp:
    asn = load_asn(fp)
```

What exactly is returned depends on what the association is. However, for all Level2 and Level3 associations, a Python `dict` (https://docs.python.org/3/library/stdtypes.html#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 Level3 assocations:

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

Since the JWST pipeline uses associations extensively, higher-level access is gained by opening an association as a JWST Data Model:

```
from jwst.datamodels import open as dm_open
container_model = dm_open('jw12345_xxxx_asn.json')
```

Utilities

Other useful utilities for creating and manipulating associations:

- `asn_from_list`
- *many other TBD*

JWST Associations

JWST Conventions

Naming Conventions

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

```
jwPPPPP-TNNNN_YYYYMMDDtHHMMSS_ATYPE_MMM_asn.json
```

where:

- `jw`: All JWST-related products begin with `jw`
- `PPPPP`: 5 digit proposal number
- `TNNNN`: Canididate 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`: Reserverd 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*
- MMM: 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 asn-DMS-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:

- image2: Intended for calwebb_image2 processing
- image3: Intended for calwebb_image3 processing
- ami3: Intended for calwebb_ami3 processing
- coron3: Intended for calwebb_coron3 processing
- tso-image2: Intended for calwebb_tso_image2 processing
- tso-spec2: Intended for calwebb_tso_spec2 processing
- tso3: Intended for calwebb_tso3 processing
- spec2: Intended for calwebb_spec2 processing
- spec3: Intended for calwebb_spec3 processing
- nrslamp-spec2: Intended for calwebb_nrslamp_spec2 processing
- wfs-image2: Intended for calwebb_wfs_image2 processing
- wfs: Wave front sensing data, used by wfs_combine

Science Data Processing Workflow

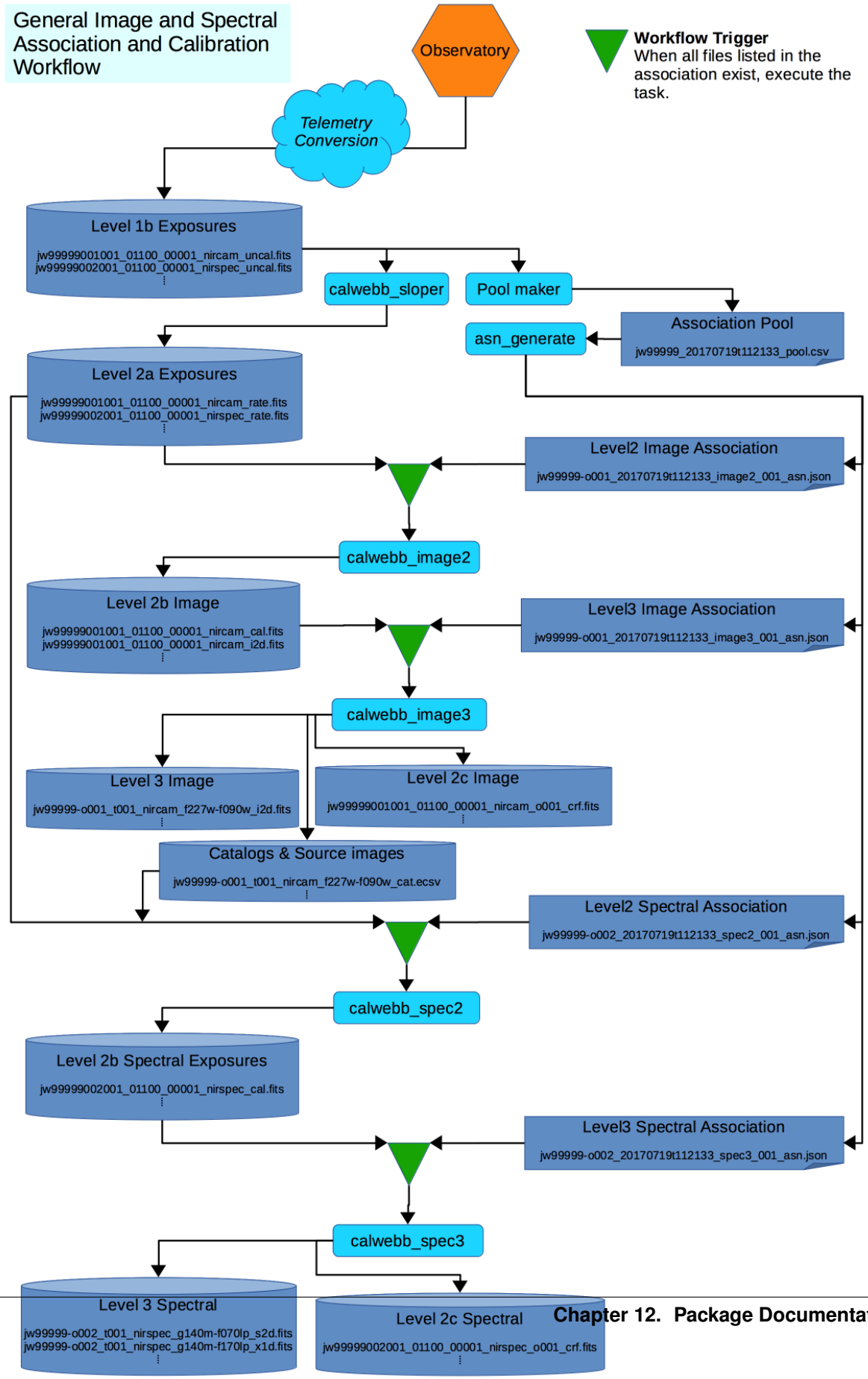
General Workflow

See level3-asn-jwst-overview for an overview of how JWST uses associations. This document describes how associations are used by the ground processing system to execute the level 2 and level 3 pipelines based on.

Up to the initial calibration step calwebb_detector1, the science exposures are treated individually. However, starting at the level 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 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 step 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:


General Image and Spectral Association and Calibration Workflow
Observatory
Workflow Trigger
When all files listed in the association exist, execute the task.
Telemetry Conversion
Level 1b Exposures
jw99999001001_01100_00001_nircam_uncal.fits
jw99999002001_01100_00001_nirspec_uncal.fits
calwebb_sloper
Pool maker
Association Pool
jw99999_20170719t112133_pool.csv
asn_generate
Level 2a Exposures
jw99999001001_01100_00001_nircam_rate.fits
jw99999002001_01100_00001_nirspec_rate.fits
Level2 Image Association
jw99999-o001_20170719t112133_image2_001_asn.json
calwebb_image2
Level 2b Image
jw99999001001_01100_00001_nircam_cal.fits
jw99999001001_01100_00001_nircam_i2d.fits
Level3 Image Association
jw99999-o001_20170719t112133_image3_001_asn.json
calwebb_image3
Level 3 Image
jw99999-o001_t001_nircam_f227w-f090w_i2d.fits
Level 2c Image
jw99999001001_01100_00001_nircam_o001_crf.fits
Catalogs & Source images
jw99999-o001_t001_nircam_f227w-f090w_cat.ecsv
Level2 Spectral Association
jw99999-o002_20170719t112133_spec2_001_asn.json
calwebb_spec2
Level 2b Spectral Exposures
jw99999002001_01100_00001_nirspec_cal.fits
Level3 Spectral Association
jw99999-o002_20170719t112133_spec3_001_asn.json
calwebb_spec3
Level 3 Spectral
jw99999-o002_t001_nirspec_g140m-f070lp_s2d.fits
jw99999-o002_t001_nirspec_g140m-f170lp_x1d.fits
Level 2c Spectral
jw99999002001_01100_00001_nirspec_o001_crf.fits

- 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 and 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 level 2 to level 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 level 3. At the level 3 stage, a source catalog of objects found in the image, and a segment map, used to record the minimum bounding box sizes for each object, are generated. The source catalog is then used as input to the level 2 processing of the spectral data. This extra link between the two major stages is represented by the `Segment & Catalog` file set, show in red in the diagram. The level 2 association `grism_spec2_asn` not only lists the needed countrate exposures, but also the catalog file produced by the level 3 image processing. Hence, the workflow knows to wait for this file before continuing the spectral processing.

Field Guide to File Names

The high-level distinctions between level 2, level 3, exposure-centric, and target-centric files can be determined by the following file patterns.

- Files produced by level 3 processing

  Any file name that matches the following regex is a file that has been produced by a level 3 pipeline:

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

- Files containing exposure-centric data

  Such data have files that match the following regex:

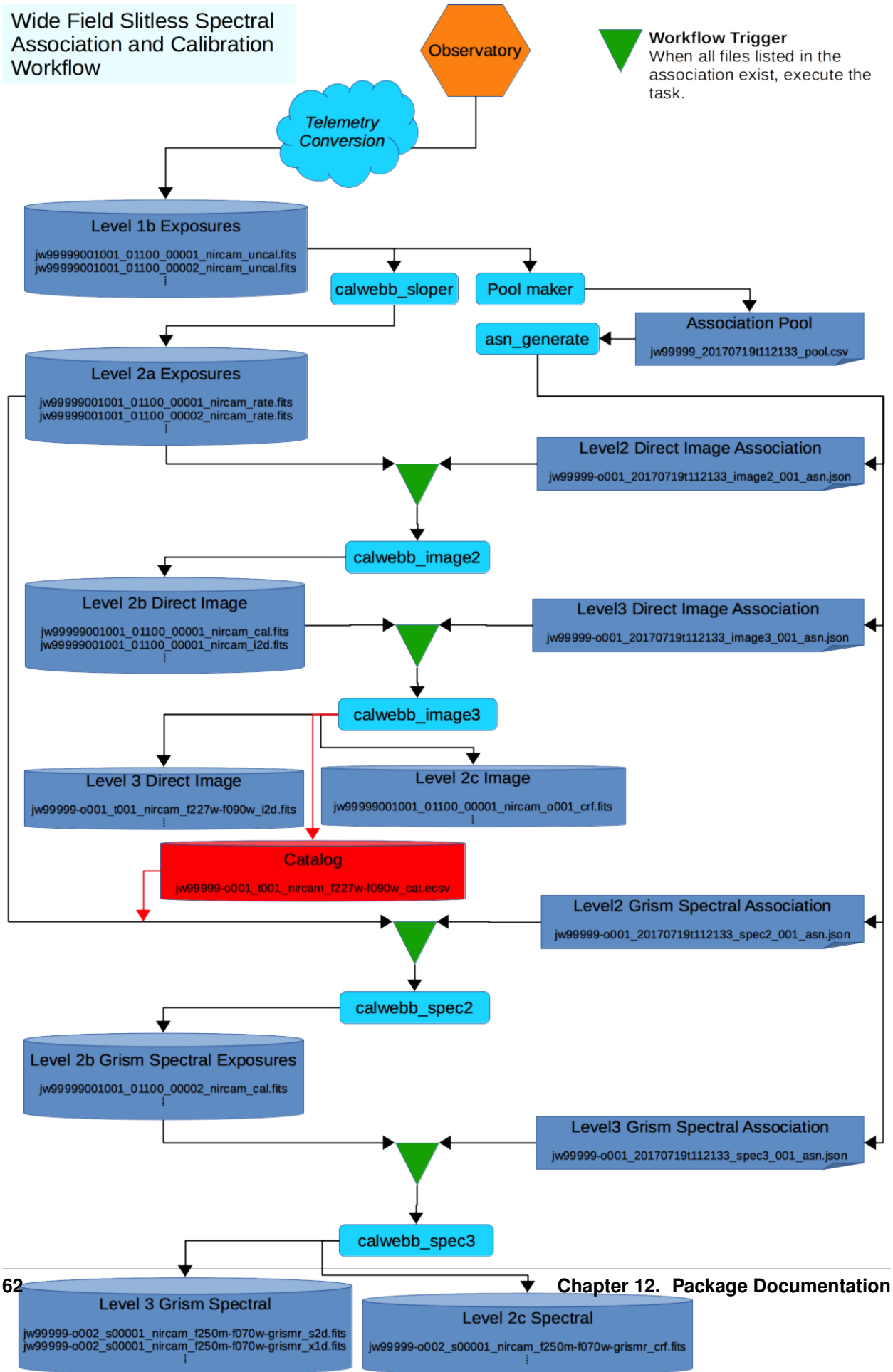
  ```
  jw[0-9]{11}_[0-9]{5}_[0-9]{5}_.+\.fits
  ```

- Files containing target-centric data

  Such data have files that match the following regex:

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

  Such data is the result of the combination of data from several exposures, usually produced by a level 3 calibration pipeline.


Wide Field Slitless Spectral Association and Calibration Workflow
Observatory
Telemetry Conversion
Workflow Trigger
When all files listed in the association exist, execute the task.
Level 1b Exposures
jw99999001001_01100_00001_nircam_uncal.fits
jw99999001001_01100_00002_nircam_uncal.fits
calwebb_sloper
Pool maker
Association Pool
jw99999_20170719t112133_pool.csv
asn_generate
Level 2a Exposures
jw99999001001_01100_00001_nircam_rate.fits
jw99999001001_01100_00002_nircam_rate.fits
Level2 Direct Image Association
jw99999-o001_20170719t112133_image2_001_asn.json
calwebb_image2
Level 2b Direct Image
jw99999001001_01100_00001_nircam_cal.fits
jw99999001001_01100_00001_nircam_i2d.fits
Level3 Direct Image Association
jw99999-o001_20170719t112133_image3_001_asn.json
calwebb_image3
Level 3 Direct Image
jw99999-o001_t001_nircam_f227w-f090w_i2d.fits
Level 2c Image
jw99999001001_01100_00001_nircam_o001_crf.fits
Catalog
jw99999-o001_t001_nircam_f227w-f090w_cat.ecsv
Level2 Grism Spectral Association
jw99999-o001_20170719t112133_spec2_001_asn.json
calwebb_spec2
Level 2b Grism Spectral Exposures
jw99999001001_01100_00002_nircam_cal.fits
Level3 Grism Spectral Association
jw99999-o001_20170719t112133_spec3_001_asn.json
calwebb_spec3
Level 3 Grism Spectral
jw99999-o002_s00001_nircam_f250m-f070w-grismr_s2d.fits
jw99999-o002_s00001_nircam_f250m-f070w-grismr_x1d.fits
Level 2c Spectral
jw99999-o002_s00001_nircam_f250m-f070w-grismr_crf.fits

Note that 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.

Level 2 Associations: Technical Specifications

Logical Structure

Independent of the actual format, all Level 2 associations have the following structure. Again, the structure is defined and enforced by the Level 2 schema

- Top level, or meta, key/values
- List of products, each consisting of
  - Output product name template
  - List of exposure members, each consisting of
    - filename of the input exposure
    - Type of exposure
    - Errors from the observatory log
    - Association Candidates this exposure belongs to

Example Association

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

```
{
    "asn_rule": "Asn_Lv2Spec",
    "asn_pool": "jw82600_001_20160304T145416_pool",
    "program": "82600",
    "asn_type": "spec2",
    "products": [
        {
            "name": "test_lrs1",
            "members": [
                {
                    "expname": "test_lrs1_rate.fits",
                    "exptype": "science"
                }
            ]
        },
        {
            "name": "test_lrs2bkg",
            "members": [
                {
                    "expname": "test_lrs2bkg_rate.fits",
                    "exptype": "science"
                }
            ]
        },
        {
            "name": "test_lrs2",
```

(continues on next page)

(continued from previous page)

```
            "members": [
                {
                    "expname": "test_lrs2_rate.fits",
                    "exptype": "science"
                },
                {
                    "expname": "test_lrs2bkg_rate.fits",
                    "exptype": "background"
                }
            ]
        }
    ]
}
```

Association Meta Keywords

The following are the top-level, or meta, keywords of an association.

program ***optional*** Program number for which this association was created.

asn_type ***optional*** The type of association represented. See level3-asn-association-types

asn_id ***optional*** The association id. The id is what appears in the asn-DMS-naming

asn_pool ***optional*** Association pool from which this association was created.

asn_rule ***optional*** Name of the association rule which created this association.

version_id ***optional*** Version identifier. DMS uses a time stamp with the format `yyyymmddthhmmss` Can be None or NULL

constraints ***optional*** List of constraints used by the association generator to create this association. Format and contents are determined by the defining rule.

`products` Keyword

A list of products that would be produced by this association. For Level2, each product is an exposure. Each product should have one `science` member, the exposure on which the Level2b processing will occur.

Association products have two components:

name ***optional*** The string template to be used by Level 2b 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 Level2b 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 Level 2b processing tasks. This keyword is explained in detail in the next section.

`members` Keyword

`members` is a list of objects, each consisting of the following keywords

expname ***required*** The exposure file name

exptype ***required*** Type of information represented by the exposure. Possible values are

- `science` *required*

  Primary science exposure. For each product, only one exposure can be `science`.

- `background` *optional*

  Off-target background exposure to subtract.

- `imprint` *optional*

  Imprint exposure to subtract.

- `sourcecat` *optional*

  The catalog of sources to extract spectra for. Usually produced by `calwebb_image3` for wide-field slitless spectroscopy.

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"
},
```

Level 3 Associations: Technical Specifications

Logical Structure

Independent of the actual format, all Level 3 associations have the following structure. Again, the structure is defined and enforced by the Level 3 schema

- Top level, or meta, key/values
- List of products, each consisting of
  - Output product name template
  - List of exposure members, each consisting of
    - filename of the input exposure
    - Type of exposure
    - Errors from the observatory log
    - 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 top-level, or meta, keywords of an association.

program ***optional*** Program number for which this association was created.

target ***optional*** Target ID for which this association refers to. DMS currently uses the TARGETID header keyword in the Level2 exposure files, but there is no formal restrictions on value.

asn_type ***optional*** The type of association represented. See level3-asn-association-types

asn_id ***optional*** The association id. The id is what appears in the asn-DMS-naming

asn_pool ***optional*** Association pool from which this association was created.

asn_rule ***optional*** Name of the association rule which created this association.

degraded_status ***optional*** Error status from the observation logs. If none the phrase "No known degraded exposures in association." is used.

version_id ***optional*** Version identifier. DMS uses a time stamp with the format `yyyymmddthhmmss` Can be None or NULL

constraints ***optional*** List of constraints used by the association generator to create this association. Format and contents are determined by the defining rule.

`products` Keyword

Association products have to components:

name ***optional*** The string template to be used by Level 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 Level3 processing modules will create a name root.

members ***required*** This is a list of the exposures to be used by the Level 3 processing tasks. This keyword is explained in detail in the next section.

`members` Keyword

`members` is a list of objects, each consisting of the following keywords

expname ***required*** The exposure file name

exptype ***required*** Type of information represented by the exposure. Possible values are

- `science` *required*

  The primary science expsoures. There is usually more than one since Level3 calibration involves combining multiple science exposures. However, at least one exposure in an association needs to be `science`.

- `psf` *optional*

  Exposures that should be considered PSF references for coronagraphic and AMI calibration.

exposerr ***optional*** If there was some issue the occured 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"
},
```

Level3 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.

- 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 Level1b, 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 identificaiton 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.

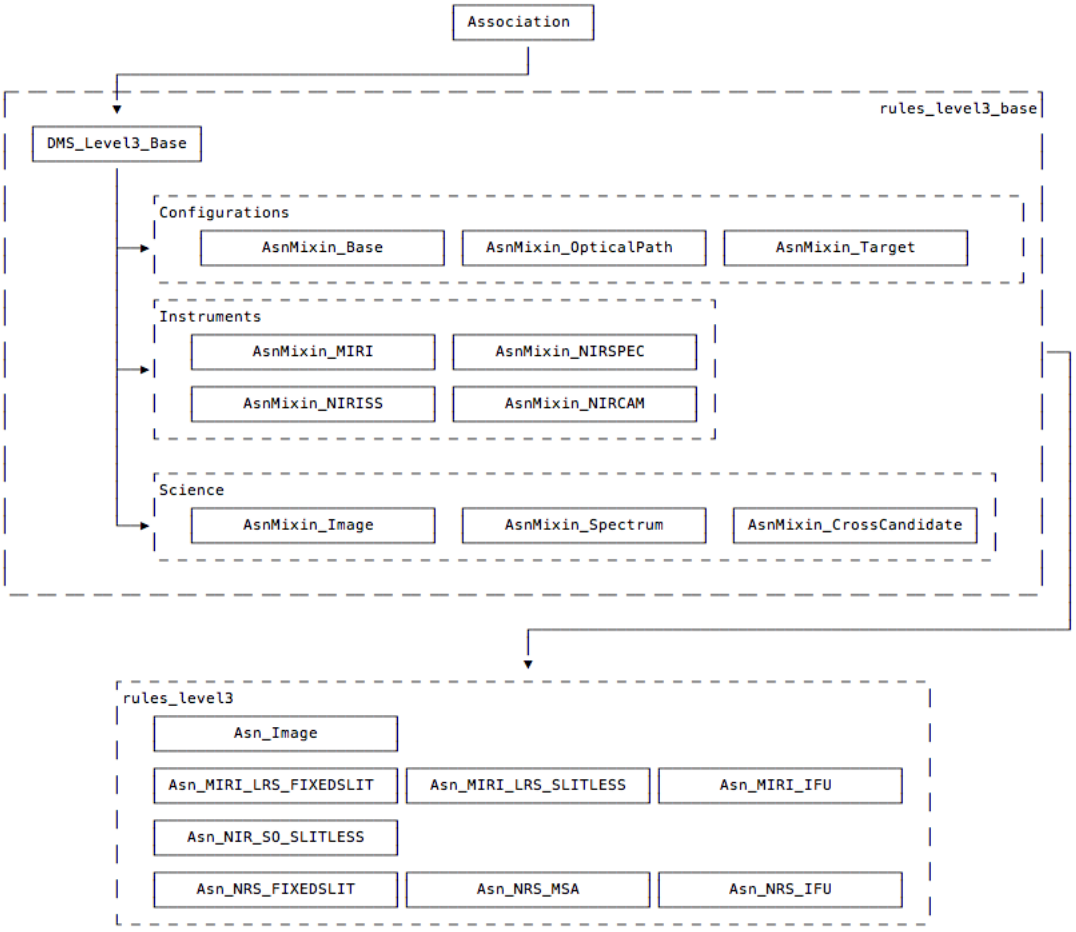


Fig. 3: Level3 Rule Class Inheritance

Level3 Rules

Association Definitions: DMS Level3 product associations

class jwst.associations.lib.rules_level3.**Asn_Image**(**args*, ***kwargs*)
Non-Association Candidate Dither Associations

class jwst.associations.lib.rules_level3.**Asn_WFSCMB**(**args*, ***kwargs*)
Wavefront Sensing association

Notes

Defined by TRAC issue #269 (https://aeon.stsci.edu/ssb/trac/jwst/ticket/269)

class jwst.associations.lib.rules_level3.**Asn_SpectralTarget**(**args*, ***kwargs*)
Slit-like, target-based, or single-object spectrographic modes

class jwst.associations.lib.rules_level3.**Asn_SpectralSource**(**args*, ***kwargs*)
Slit-like, multi-object spectrographic modes

dms_product_name
Define product name.

Returns **product_name** – The product name

Return type str (https://docs.python.org/3/library/stdtypes.html#str)

class jwst.associations.lib.rules_level3.**Asn_IFU**(**args*, ***kwargs*)
IFU associations

dms_product_name
Define product name.

class jwst.associations.lib.rules_level3.**Asn_Coron**(**args*, ***kwargs*)
Coronography .. rubric:: Notes

Coronography is nearly completely defined by the association candidates produced by APT. Tracking Issues: - github #311

class jwst.associations.lib.rules_level3.**Asn_AMI**(**args*, ***kwargs*)
Aperture Mask Interferometry .. rubric:: Notes

AMI is nearly completely defined by the association candidates produced by APT. Tracking Issues: - github #310

class jwst.associations.lib.rules_level3.**Asn_WFSS_NIS**(**args*, ***kwargs*)
WFSS/Grism modes

dms_product_name
Define product name.

Returns **product_name** – The product name

Return type str (https://docs.python.org/3/library/stdtypes.html#str)

class jwst.associations.lib.rules_level3.**Asn_TSO**(**args*, ***kwargs*)
Time-Series observations

Design

Association Design

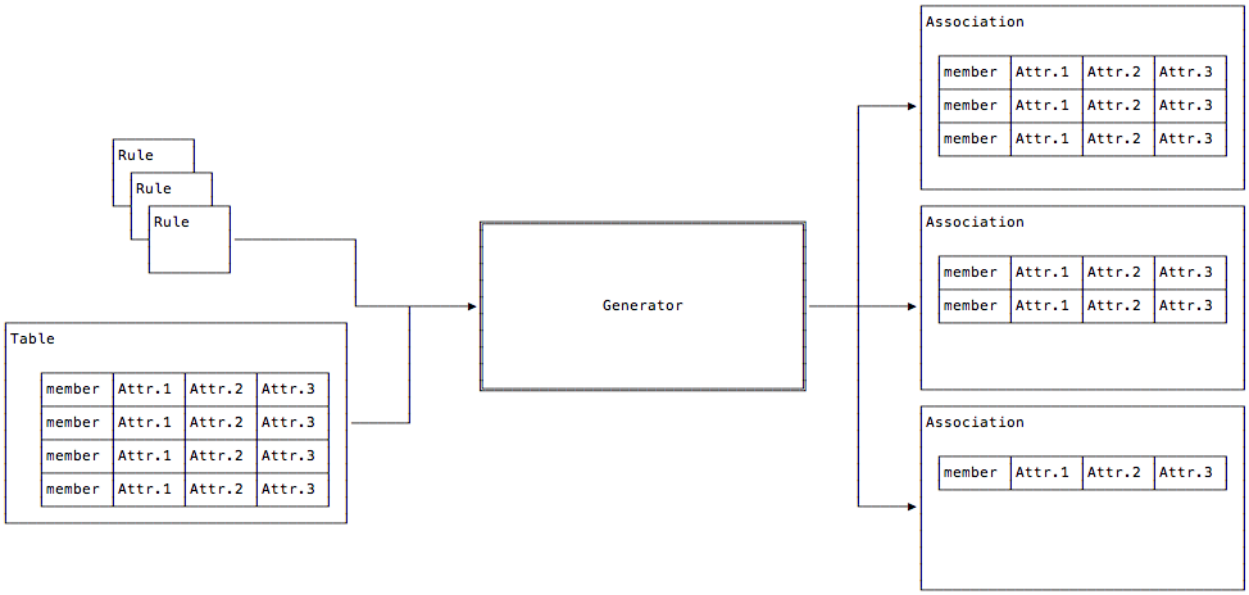


Fig. 4: Association Generator Overview

As introduced in the 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. 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 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.

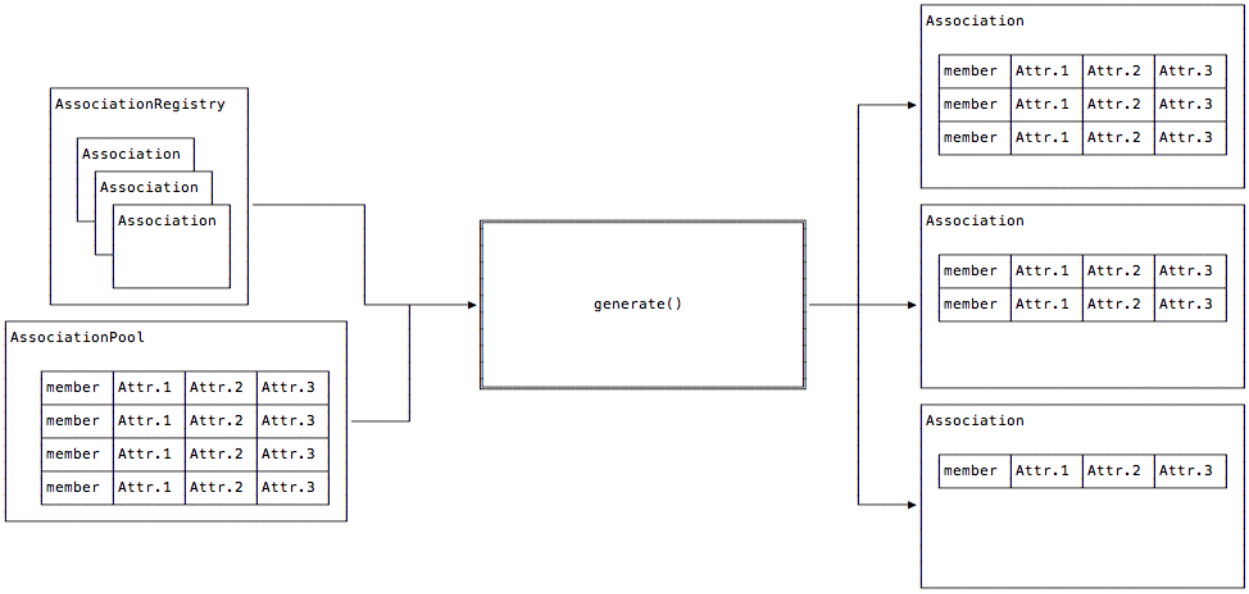


Fig. 5: Association Class Relationship overview

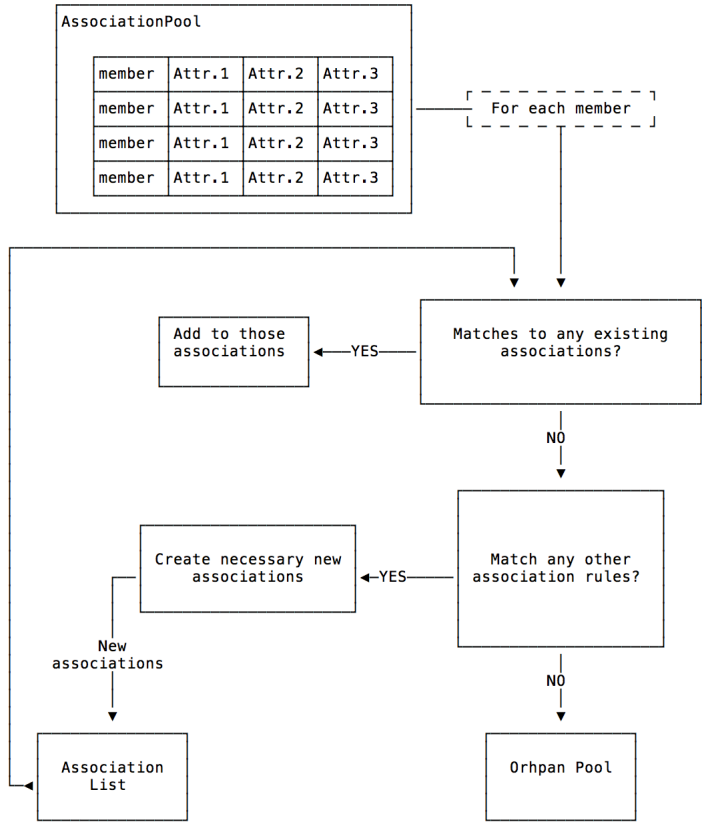


Fig. 6: Generator Conceptual Workflow

Output

Before exiting, the generate checks the `Association.is_valid` property of each association to ensure that an association has all the members it is required to have. With respect to JWST and Level3 processing, an example of an association that would not be valid would be if an observation failed to complete, producing only a subset of exposures. The result would be an invalid association, since any further processing would fail.

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.

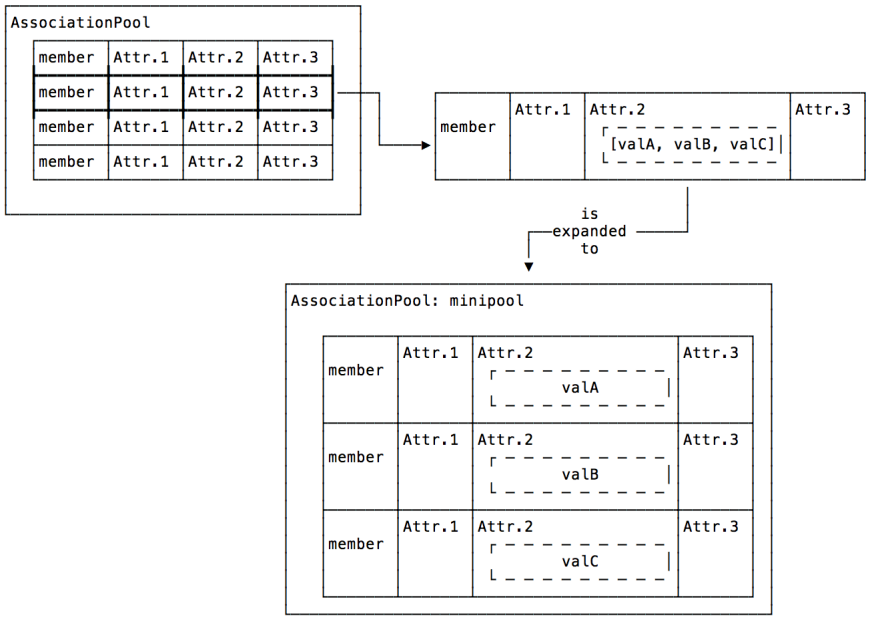


Fig. 7: Member list expansion
Attr.2 is a list of three values which expands into three members in the mini pool.

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:

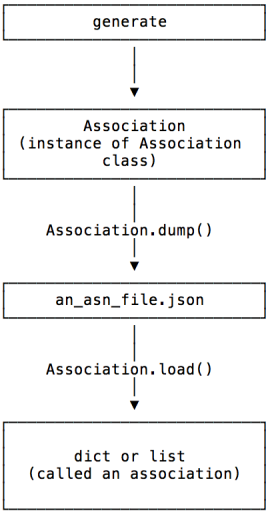


Fig. 8: Rule vs. Association Relationship

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 Level2-associations and Level3-associations 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.

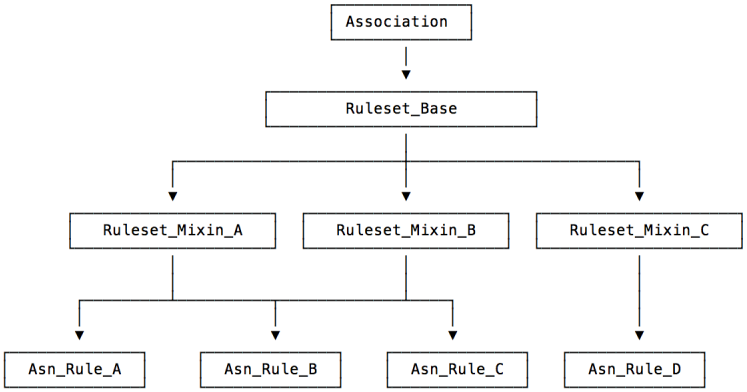


Fig. 9: Rule Inheritance

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 *`jwst.associations.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

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 and the orphaned exposure table.

asn_from_list

Create an association using either the command line tool `asn_from_list` or through the Python API using either `jwst.associatons.asn_from_list.Main` or `jwst.associations.asn_from_list.asn_from_list()`

Command Line

```
asn_from_list --help
```

Usage

Level2 Associations

Refer to *Level 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 the 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": "jw99999001001_011001_00001_nirspec_rate.fits",
                "exptype": "science"
            },
            {
                "expname": "jw99999001001_011001_00002_nirspec_rate.fits",
                "exptype": "background"
            },
            {
                "expname": "jw99999001001_011001_00003_nirspec_rate.fits",
                "exptype": "imprint"
            }
        ]
    }
]
```

Level3 Associations

Refer to *Level 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 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": "jw99999001001_011001_00001_nircam_cal.fits",
                "exptype": "science"
            },
            {
                "expname": "jw99999001001_011001_00002_nircam_cal.fits",
                "exptype": "science"
            },
            {
                "expname": "jw99999001001_011001_00003_nircam_cal.fits",
                "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.

Association Rules

Association definitions, or `rules`, are Python classes, all based on the 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 level3-associations, 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` (https://docs.python.org/3/library/constants.html#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` (https://docs.python.org/3/library/constants.html#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` (https://docs.python.org/3/library/constants.html#None) as the new association.

Any override of this method is expected to first call `super` (https://docs.python.org/3/library/functions.html#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 structured 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

Setup default and environment

Functions

generate(pool, rules[, version_id])	Generate associations in the pool according to the rules.
generate_from_item(item, version_id, ...)	Either match or generate a new assocation
libpath(filepath)	Return the full path to the module library.
load_asn(serialized[, format, first, ...])	Load an Association from a file or object

Continued on next page

Table 15 – continued from previous page

make_timestamp()	
match_item(item, associations)	Match item to a list of associations

generate

jwst.associations.**generate**(*pool*, *rules*, *version_id=None*)

Generate associations in the pool according to the rules.

Parameters

- **pool** (*AssociationPool*) – The pool to generate from.
- **rules** (*Associations*) – The associaton rule set.
- **version_id** (*None* (https://docs.python.org/3/library/constants.html#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.

Returns **associations** – List of associations

Return type [association[,..]]

Notes

Refer to the Association Generator documentation for a full description.

generate_from_item

jwst.associations.**generate_from_item**(*item*, *version_id*, *associations*, *rules*, *process_list*)

Either match or generate a new assocation

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to match to existing associations or generate new associations from
- **version_id** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None* (https://docs.python.org/3/library/constants.html#None)) – Version id to use with association creation. If None, no versioning is used.
- **associations** (*[association, ..]*) – List of already existing associations. If the item matches any of these, it will be added to them.
- **rules** (*AssociationRegistry or None* (https://docs.python.org/3/library/constants.html#None)) – List of rules to create new associations
- **process_list** (*ProcessList*) – The *ProcessList* from which the current item belongs to.

Returns

(associations, process_list) –

existing_asns: [association,...] List of existing associations item belongs to. Empty if none match

new_asns: [association,...] List of new associations item creates. Empty if none match

process_list: [ProcessList, ...] List of process events.

Return type 3-tuple where

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* (https://docs.python.org/3/library/constants.html#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* (https://docs.python.org/3/library/constants.html#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`

Returns

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.

make_timestamp

jwst.associations.**make_timestamp**()

match_item

jwst.associations.**match_item**(*item*, *associations*)
Match item to a list of associations

Parameters

- **item** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – The item to match to the associations.
- **associations** (*[association, ..]*) – List of already existing associations. If the item matches any of these, it will be added to them.

Returns

(associations, process_list) –

associations: [association,...] List of associations item belongs to. Empty if none match

process_list: [ProcessList, ...] List of process events.

Return type 2-tuple where

Classes

Association([version_id])	Association Base Class
AssociationError([message])	Basic errors related to Associations
AssociationNotAConstraint([message])	No matching constraint found
AssociationNotValidError([message])	Given data structure is not a valid association
AssociationPool([data, masked, names, ...])	Association Pool
AssociationRegistry([definition_files, ...])	The available assocations
ProcessList([items, rules, work_over, ...])	A Process list
ProcessQueueSorted([init])	Sort ProcessItem based on work_over
RegistryMarker	Mark rules, callbacks, and module

Association

class jwst.associations.**Association**(*version_id=None*)
Bases: collections.abc.MutableMapping (https://docs.python.org/3/library/collections.abc.html#collections.abc.Mutab

Association Base Class

Parameters **version_id** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None* (https://docs.python.org/3/library/constants.html#None)) – Version_Id to use in the name of this association. If None, nothing is added.

Raises *AssociationError* – If a item doesn't match any of the registered associations.

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 assocations and, if defined, valided 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)

registry
The registry this association came from.

Type *AssociationRegistry*

asn_name
The suggested file name of association

Type str (https://docs.python.org/3/library/stdtypes.html#str)

asn_rule
The name of the rule

Type str (https://docs.python.org/3/library/stdtypes.html#str)

AssociationError

exception jwst.associations.**AssociationError**(*message='No explanation given'*)
Basic errors related to Associations

AssociationNotAConstraint

exception jwst.associations.**AssociationNotAConstraint**(*message='No explanation given'*)
No matching constraint found

AssociationNotValidError

exception jwst.associations.**AssociationNotValidError**(*message='No explanation given'*)
Given data structure is not a valid association

AssociationPool

class jwst.associations.**AssociationPool**(*data=None*, *masked=None*, *names=None*, *dtype=None*, *meta=None*, *copy=True*, *rows=None*, *copy_indices=True*, ***kwargs*)
Bases: astropy.table.table.Table

Association Pool

An AssociationPool is essentially and astropy Table with the following default behaviors:

- ASCII tables with a default delimiater 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

write(**args*, ***kwargs*)
Write the pool to a file.

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 assocations

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.

rule_set
The rules in the registry.

Type {rule [, ..]}

match(*item*)
Return associations where `item` matches any of the rules.

validate(*association*)
Determine whether an association is valid, or complete, according to any of the rules in the registry.

finalize(*associations*)
Validate and execute post-processing hooks to produce a completed and valid set of associations.

load(*serialized*)
Create an association from a serialized form.

Notes

The general workflow is as follows:

- **Create the registry**

```
>>> registry = AssociationRegistry()
```

- **Create associations from an item**

```
>>> associations, reprocess = registry.match(item)
```

- **Finalize the associations**

```
>>> final_asns = registry.finalize(assocations)
```

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 a full example of using the registry.

Attributes Summary

rule_set	

Methods Summary

add_rule(name, obj[, global_constraints])	Add object as rule to registry
load(serialized[, format, validate, first])	Marshall 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
validate(association)	Validate a given association against schema

Attributes Documentation

rule_set

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*)
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* (https://docs.python.org/3/library/constants.html#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

Returns

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)) – A item, like from a Pool, to find assocations for.
- **version_id** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – If specified, a string appened 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

Parameters

- **module** (*module*) – The module, and all submodules, to be parsed.
- **Modifies** –
- **--------** –
- **self.callback** – Found callbacks are added to the callback registry

validate(*association*)

Validate a given association against schema

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

ProcessList

class jwst.associations.**ProcessList**(*items=None*, *rules=None*, *work_over=1*, *only_on_match=False*)

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
- **only_on_match** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Only use this object if the overall condition is True.

Attributes Summary

BOTH
EXISTING
NONSCIENCE
RULES

Attributes Documentation

BOTH = 1

EXISTING = 2

NONSCIENCE = 3

RULES = 0

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 module

Methods Summary

callback(event)	Mark object as a callback for an event
is_marked(obj)	
mark(obj)	Mark that 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.
- **Modifies** –
- **--------** –
- **_asnreg_role** (*'callback'*) – Attributed added to object and set to *rule*
- **_asnreg_events** (*[event[, ..]]*) – The events this callable object is a callback for.
- **_asnreg_mark** (*True*) – Attributed added to object and set to True

Returns **obj** – Return object to enable use as a decorator.

Return type object (https://docs.python.org/3/library/functions.html#object)

static is_marked(*obj*)

static mark(*obj*)
Mark that object should be part of the registry

Parameters

- **obj** (*object* (https://docs.python.org/3/library/functions.html#object)) – The object to mark
- **Modifies** –
- **--------** –
- **_asnreg_mark** (*True*) – Attribute added to object and is set to True
- **_asnreg_role** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or None* (https://docs.python.org/3/library/constants.html#None)) – Attribute added to object indicating role this object plays. If None, no particular role is indicated.

Returns **obj** – Return object to enable use as a decorator.

Return type object (https://docs.python.org/3/library/functions.html#object)

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
- **Modifies** –
- **--------** –
- **_asnreg_role** (*'rule'*) – Attributed added to object and set to *rule*
- **_asnreg_mark** (*True*) – Attributed added to object and set to True

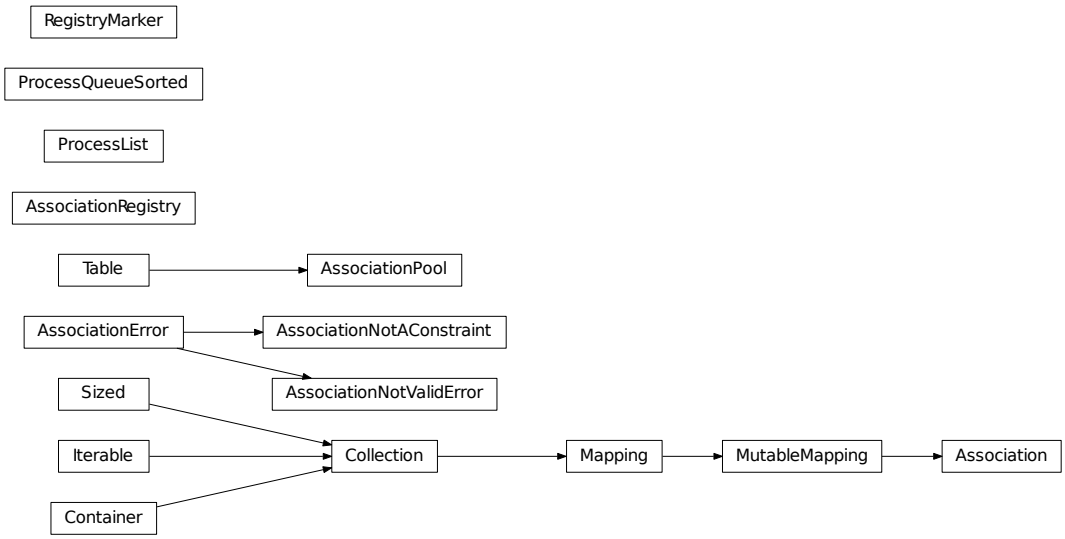
Returns **obj** – Return object to enable use as a decorator.

Return type object (https://docs.python.org/3/library/functions.html#object)

static schema(*filename*)
Mark a file as a schema source

static utility(*class_obj*)
Mark the class as a Utility class

Class Inheritance Diagram



12.1.5 Background Image Subtraction

Description

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).

Non-WFSS 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.

The average background image is produced as follows:

- Clip the combined SCI arrays of all background exposures
- Compute the mean of the unclipped SCI values
- 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

- Combine the DQ arrays of all background exposures using a bitwise-OR operation

The average background exposure is then subtracted from the target exposure. The subtraction consists of the following operations:

- The SCI array of the average background is subtracted from the SCI array of the target exposure
- The ERR array of the target exposure is currently unchanged, until full error propagation is implemented in the entire pipeline
- 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 background image is subtracted from each plane of the CubeModel.

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 target data 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 relavant to the target data. All regions of the image that are free of source spectra are used for scaling the background reference image. Robust mean values are obtained for the background regions in the target 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 two optional arguments, both of which 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` The number of standard deviations to use for the clipping limit. Defaults to 3.

`--maxiters` The number of clipping iterations to perform, or `None` to clip until convergence is achieved. 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.

WFSS Background reference file

REFTYPE WFSSBKG

Data model `WfssBkgModel`

The WFSS background reference file contains a "master" image of the dispersed background produced by a particular filter+grism combination.

CRDS Selection Criteria

WFSSBKG reference files are selected by:

INSTRUME, DETECTOR, EXP_TYPE, FILTER, and PUPIL

Required Keywords

The following table lists the keywords that are required to be present in a WFSSBKG reference file. An asterisk following a keyword name indicates a standard keyword that is required in all reference files, regardless of type.

Keyword	Model Name
AUTHOR*	meta.author
DATAMODL*	meta.model_type
DATE*	meta.date
DESCRIP*	meta.description
DETECTOR	meta.instrument.detector
EXP_TYPE	meta.exposure.type
FILENAME*	meta.filename
FILTER	meta.instrument.filter
INSTRUME*	meta.instrument.name
PEDIGREE*	meta.pedigree
PUPIL	meta.instrument.pupil
REFTYPE*	meta.reftype
TELESCOP*	meta.telescope
USEAFTER*	meta.useafter

Reference File Format

WFSSBKG 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 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

Wavelength Range reference file

REFTYPE WAVELENGTHRANGE

Data model `WavelengthrangeModel`

The wavelength range reference file contains information about the range of wavelengths in the exposure. It is used, together with a source catalog, to create a mask giving the locations of source spectra in the target image and hence where the background regions are.

CRDS Selection Criteria

Wavelengthrange reference files are selected by:

INSTRUME, EXP_TYPE, PUPIL (NIRCam only), and MODULE (NIRCam only)

jwst.background Package

Classes

SubtractImagesStep([name, parent, ...])	SubtractImagesStep: Subtract two exposures from one another to accomplish background subtraction.
BackgroundStep([name, parent, config_file, ...])	BackgroundStep: Subtract background exposures from target exposures.

SubtractImagesStep

class jwst.background.**SubtractImagesStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

SubtractImagesStep: Subtract two exposures from one another to accomplish background subtraction.

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.

- ****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

spec	

Methods Summary

process(input1, input2)	Subtract the background signal from a JWST data model by subtracting a background image from it.

Attributes Documentation

spec = '\n '

Methods Documentation

process(*input1*, *input2*)

Subtract the background signal from a JWST data model by subtracting a background image from it.

Parameters

- **input1** (*JWST data model*) – input science data model to be background-subtracted
- **input2** (*JWST data model*) – background data model

Returns **result** – background-subtracted science data model

Return type JWST data model

BackgroundStep

class jwst.background.**BackgroundStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

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.
- ****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

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

reference_file_types = ['wfssbkg', 'wavelengthrange']

spec = '\n sigma = float(default=3.0) # Clipping threshold\n maxiters = integer(defaul

Methods Documentation

process (*input*, *bkg_list*)

Subtract the background signal from target exposures by subtracting designated background images from them.

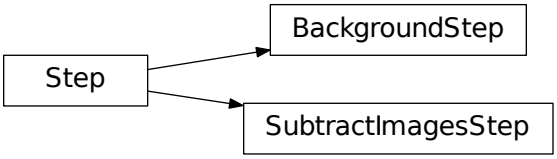
Parameters

- **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



12.1.6 Barshadow Correction

Description

Overview

The `barshadow` step calculates the correction to be applied to NIRSpec MSA data for uniform 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 calspec2 pipeline.

Input details

The input data should be from after the extract_2d step, so that it contains cutouts around each slitlet.

Algorithm

The 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.

Output product

The output product has the barshadow correction attached to each slit of the multislit datamodel in the BARSHADOW extension.

Reference File

The barshadow step does uses the barshadow reference file.

CRDS Selection Criteria

The Barshadow reference file is selected only for exposures with EXP_TYPE=NRS_MSASPEC. All other EXP_TYPEs should return N/A.

Reference File Format

The barshadow reference file is a FITS file that contains four extensions:

EXTNAME	NAXIS	Dimensions	Data type
DATA1X1	2	101x1001	float
VAR1X1	2	101x1001	float
DATA1X3	2	101x1001	float
VAR1X3	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 0.6x10^-6m to 5.3x10^-6m.

The extension headers look like this:

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		

Step Arguments

The barshadow step has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

reference_file_types = ['barshadow']

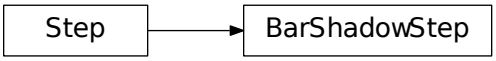
spec = '\n '

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



12.1.7 Combine 1D Spectra

Description

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. It is the net count rate that uses this weight; that is, the net count rate is multiplied by the integration time to get net counts, and it is the net counts that are added together and finally divided by the sum of the integration times. The flux weighted by an additional factor of the instrumental sensitivity, count rate per unit flux. The idea is that the quantity that is added up should be in units of counts. If unit weight was specified, however, unit weight will be used for both flux and net. The data quality (DQ) columns will be combined using bitwise OR.

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. An input data file can be either SpecModel (for one spectrum) or 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 six columns, giving the wavelength, flux, error estimate for the flux, net countrate in counts/second, 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.

Reference File

This step does not use any reference file.

Step Arguments

The combine_1d step has two step-specific arguments:

- `--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.

- `--interpolation`

This is a string that specifies how to interpolate between pixels of the input data. The default value is "nearest", which means that no actual interpolation will be done; the pixel number will be rounded to an integer, and the input value at that pixel will be used.

This argument is not currently used. It is included as a placeholder for a possible future enhancement.

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: *jwst.stpipe.Step*

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.
- ****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

spec	

Methods Summary

process(input_file)	This is where real work happens.

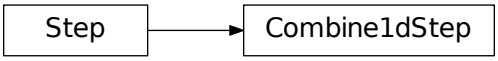
Attributes Documentation

spec = '\n # integration_time or exposure_time.\n exptime_key = string(default="integra

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



12.1.8 Coronagraphic Processsing

Tasks in the package

The coronagraphic package currently consists of the following tasks:

- stack_refs
- align_refs
- klip
- hlsp

Briefly, the `stack_refs` step is used to load images of reference PSF targets, as listed in an Association file, and stack the images into a data cube in a single file to be used in subsequent processing steps. The `align_refs` step is then used to align the stacked reference PSF images with the images contained in a science target exposure. The `klip` step applies the Karhunen-Loeve Image Plane (KLIP) algorithm to the aligned reference PSF and science target images and produces PSF-subtracted science target images. The `hlsp` task produces high-level science products (HLSP's) from a KLIP-subtracted image.

CALWEBB_CORON3

Currently the individual steps can only be run in a convenient way by running the `calwebb_coron3` pipeline, which calls the individual steps and takes care of all the necessary loading and passing of data models for the input and output products of each step. The input to the `calwebb_coron3` pipeline is expected to be an ASN file. The ASN file should define a single output product, which will be the combined image formed from the PSF-subtracted results of all the input science target data. That output product should then define, as its members, the various input reference PSF and science target files to be used in the processing. An example ASN file is shown below.

```
{"asn_rule": "CORON", "target": "NGC-3603", "asn_pool": "jw00017_001_01_pool",
↪"program": "00017",
"products": [
    {"prodtype": "coroncmb", "name": "jw89001-c1001_t001_nircam_f160w",
     "members": [
         {"exptype": "science", "expname": "test_targ1_calints.fits"},
         {"exptype": "science", "expname": "test_targ2_calints.fits"},
         {"exptype": "psf", "expname": "test_psf1_calints.fits"},
         {"exptype": "psf", "expname": "test_psf2_calints.fits"},
         {"exptype": "psf", "expname": "test_psf3_calints.fits"}]}],
"asn_type": "coron",
"asn_id": "c1001"}
```

In this example the output product "jw89001-c1001_t001_nircam_f160w" is defined to consist of 2 science target inputs and 3 reference psf inputs. Note that the values of the `exptype` attribute for each member are very important and used by the `calwebb_coron3` pipeline to know which members are to be used as reference PSF data and which are data for the science target. The output product name listed in the ASN file is used as the root name for some of the products created by the `calwebb_coron3` pipeline. This includes:

- rootname_psfstack: the output of the `stack_refs` step
- rootname_i2d: the final combined target image

Other products will be created for each individual science target member, in which case the root names of the original input science target products will be used as a basis for the output products. These products include:

- targetname_psfalign: the output of the `align_refs` step
- targetname_psfsub: the output of the `klip` step

Stack_refs

Overview

The `stack_refs` step takes a list of reference PSF products and stacks all of the images in the PSF products into a single 3D data cube. It is assumed that the reference PSF products are in the form of a data cube (jwst CubeModel type data model) to begin with, in which images from individual integrations are stacked along the 3rd axis of the data cube. Each data cube from an input reference PSF file will be appended to a new output 3D data cube (again a CubeModel), such that the dimension of the 3rd axis of the output data cube will be equal to the total number of integrations contained in all of the input files.

Inputs and Outputs

The `stack_refs` step is called from the `calwebb_coron3` pipeline module. The `calwebb_coron3` pipeline will find all of the `psf` members listed in the input ASN file, load each one into a CubeModel data model, and construct a ModelContainer that is the list of all psf CubeModels. The ModelContainer is passed as input to the `stack_refs`

step. The output of `stack_refs` will be a single CubeModel containing all of the concatenated data cubes from the input psf 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: *jwst.stpipe.Step*

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.
- ****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

spec	

Methods Summary

process(input)	This is where real work happens.

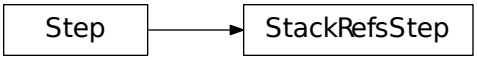
Attributes Documentation

spec = '\n '

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



Align_refs

Overview

The `align_refs` step is used to compute offsets between science target images and the reference PSF images and shift the PSF images into alignment. Each integration contained in the stacked PSF data is aligned to each integration within a given science target product. The `calwebb_coron3` pipeline applies the `align_refs` step to each input science target product individually, resulting in a set of PSF images that are aligned to the images in that science target product.

Inputs and Outputs

The `align_refs` step takes 2 inputs: a science target product, in the form of a CubeModel data model, and the stacked PSF product, also in the form of a CubeModel data model. The resulting output is a 4D data model (QuadModel), 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.

jwst.coron.align_refs_step Module

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types	
spec	

Methods Summary

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

Attributes Documentation

reference_file_types = ['psfmask']

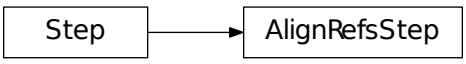
spec = '\n '

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



Klip

Overview

The `klip` task applies the KLIP algorithm to coronagraphic images, using an accompanying set of reference PSF images, in order to fit and subtract an optimal PSF from the source. 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 LOCI. The main advantages of KLIP over LOCI is the possibility of direct forward modeling and a significant speed increase.

The KLIP algorithm consists of the following steps:

1) Partition the target and reference 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

Inputs and Outputs

The `klip` task takes two inputs: a science target product, in the form of a 3D CubeModel data model, and a set of aligned PSF images, in the form of a 4D QuadModel data model. Each 'layer' in the 4th dimension of the PSF data contains all of the aligned PSF images corresponding to a given integration (3rd dimension) in the science target cube. The output from the klip step is a 3D CubeModel data model, having the same dimensions as the input science target product, and contains the PSF-subtracted images for every integration of the science target product.

Arguments

The task takes one optional argument, `truncate`, which is used to specify the number of KL transform rows to keep when computing the PSF fit to the target. The default value is 50.

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: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

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

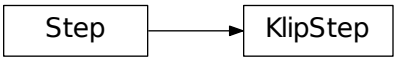
Attributes Documentation

spec = '\n truncate = integer(default=50,min=0) # The number of KL transform rows to k

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



HLSP

Overview

The `hlsp` task produces high-level science products for KLIP-processed images. The task currently produces two such 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.

Input Arguments

The `hlsp` task takes one input file name argument, which is the name of the KLIP-processed target product to be analyzed. One optional argument is available, `annuli_width`, which specifies the width (in pixels) of the annuli to use in calculating the contrast data. The default value is 2 pixels.

Outputs

The `hslp` task produces two output products. The first is the snr image (file name suffix "_snr") and the second is the table of contrast data (file name suffix "_contrast"). The contrast data are stored as a 2-column table giving radius (in pixels) and noise (1-sigma).

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: *jwst.stpipe.Step*

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.
- ****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

spec	

Methods Summary

process(target)	This is where real work happens.

Attributes Documentation

spec = '\n annuli_width = integer(default=2, min=1) # Width of contrast annuli\n save_

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

jwst.coron Package

Classes

StackRefsStep([name, parent, config_file, ...])	StackRefsStep: Stack multiple PSF reference exposures into a single CubeModel, for use by subsequent coronagraphic steps.
AlignRefsStep([name, parent, config_file, ...])	AlignRefsStep: Align coronagraphic PSF images with science target images.
KlipStep([name, parent, config_file, ...])	KlipStep: Performs KLIP processing on a science target coronagraphic exposure.

Continued on next page

Table 46 – continued from previous page

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.

StackRefsStep

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

Bases: *jwst.stpipe.Step*

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.
- ****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

spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

spec = '\n '

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.

AlignRefsStep

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

Bases: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

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

Attributes Documentation

reference_file_types = ['psfmask']

spec = '\n '

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.

KlipStep

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

Bases: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

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

Attributes Documentation

spec = '\n truncate = integer(default=50,min=0) # The number of KL transform rows to k

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.

HlspStep

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

Bases: *jwst.stpipe.Step*

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.
- ****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

spec	

Methods Summary

process(target)	This is where real work happens.

Attributes Documentation

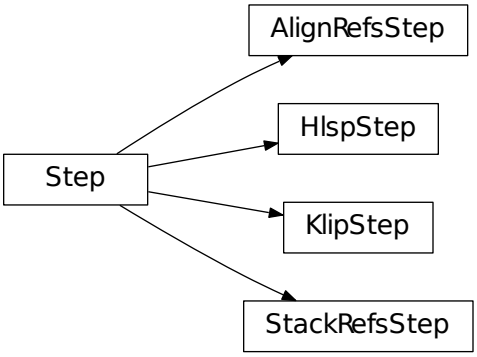
spec = '\n annuli_width = integer(default=2, min=1) # Width of contrast annuli\n save_

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



12.1.9 CSV Tools

CSV TOOLS

TBD

csvconvert

Command-line script to convert CSV files to JSON, or FITS

```
csvconvert --help
```

jwst.csv_tools Package

12.1.10 Cube Building

Description

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:

- a single file containing a 2-D slice image
- a data model (IFUImageModel) containing a 2-D slice image
- an association table (in json format) containing a list of input files
- a model container with several 2-D slice data models

There are a number of arguments the user can provide either in a configuration 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 countrate to surface brightness. This step will only work with MIRI or NIRSpec IFU data.

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 IFU's providing four 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 IFU's is split into two paths. One path is dedicated to the two short wavelength IFU's and the other one handles the two longer wavelength IFU's. 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 bands. Each slice is dispersed by a prism or one of six diffraction gratings. When using diffraction gratings as dispersive elements, three separate gratings are employed in combination with specific filters in order to avoid the overlapping of spectra caused by different grating orders. The three gratings span four partially overlapping bands (1.0 - 1.8 microns; 1.7 - 3.0 microns; 2.9 - 5 microns) covering the total spectral range in four separate exposures. Six gratings provide high-resolution (R = 1400-3600) and medium resolution (R = 500-1300) spectroscopy over the wavelength range 0.7-5 microns, while the prism yields lower-resolution (R = 30-300) spectroscopy over the range 0.6-5 microns.

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. The light-sensitive portions of the two SCAs are separated by a physical gap of 3.144 mm, which corresponds to 17.8 arcseconds on the sky. 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. The physical gap between the SCAs causes a loss of spectral information over a range in wavelength that depends on the location of the target and dispersive element used. The lost information can be recovered by dithering the targets.

Terminology

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).

NIRSpec IFU Disperser and Filter Combinations

Grating	Filter	Wavelength (microns)
Prism	Clear	0.6 -5.3
G140M	F070LP	0.7 - 1.2
G140M	F100LP	1 - 1.8
G235M	F170LP	1.7 - 3.1
G395M	F290LP	2.9 - 5.2
G140H	F070LP	0.7 - 1.2
G140H	F100LP	1 - 1.8
G235H	F170LP	1.7 - 3.1
G395H	F290LP	2.9 - 5.2

For NIRSpec we define a *band* as a single grating-filter combination, e.g. G140M-F070LP.

Coordinate Systems

An IFU spectrograph measures the intensity of a region of the sky as a function of wavelength. There are a number of different coordinate systems used in the cube building process. Here is an overview of these coordinate systems:

Detector System Defined by the hardware and presents raw detector pixel values. Each detector or SCA will have its own pixel-based coordinate system. In the case of MIRI we have two detector systems because the MIRI IFUs disperse data onto two detectors.

Telescope (V2,V3) The V2,V3 coordinates locate points on a spherical coordinate system. The frame is tied to the JWST focal plane and applies to the whole field of view, encompassing all the instruments. The V2,V3 coordinates are Euler angles in a spherical frame rather than Cartesian coordinates.

XAN,YAN Similar to V2,V3, but flipped and shifted so the origin lies between the NIRCam detectors instead of at the telescope boresight. Note that what OSIM and OTE call 'V2,V3' are actually XAN,YAN.

Absolute The standard astronomical equatorial RA/Dec system.

Cube A three-dimensional system with two spatial axes and one spectral axis.

MRS-FOV A MIRI-specific system that is the angular coordinate system attached to the FOV of each MRS band. There are twelve MRS-FOV systems for MIRI, because there are twelve bands (1A, 1B, 1C,... 4C). Each system has two orthogonal axes, one parallel (**alpha**) and the other perpendicular (**beta**) to the projection of the long axes of the slices in the FOV.

Types of Output Cubes

As mentioned above, 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. For NIRSpec this means using data from the single grating+filter combination contained in the exposure.

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-band cubes. For MIRI, for example, this can mean separate cubes for bands 1A, 2A, 3A, 4A, 1B, 2B, ..., 3C, 4C, 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.

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. 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/arcsecond^2. 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 point cloud elements contained in the region of interest of the spaxel.

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 default IFU Cubes contain data from a single band (channel/sub-channel or grating/filter). There are several options which control the type of cubes to create (see description given above). Based on the 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 included input data. This 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 'cubepars' reference file as a function of wavelength, and can also be changed by the user. The cubepars 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 output spatial coordinate system is right ascension-declination.

All the pixels on each exposure that are included are mapped to the cube coordinate system. This input-to-output pixel mapping is determined via a mapping function 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 location to its projected location in the cube coordinate system. The mapping is actually a series of chained transformations (detector -> alpha-beta-lambda), (alpha-beta-lambda -> v2-v3-lambda), (v2-v3-lambda - > right ascension-declination-lambda), and (right ascension-declination-lambda -> Cube coordinate1-Cube Coordinate2-lambda). The reverse of each transformation is also possible.

The mapping process results in an irregular 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 1.

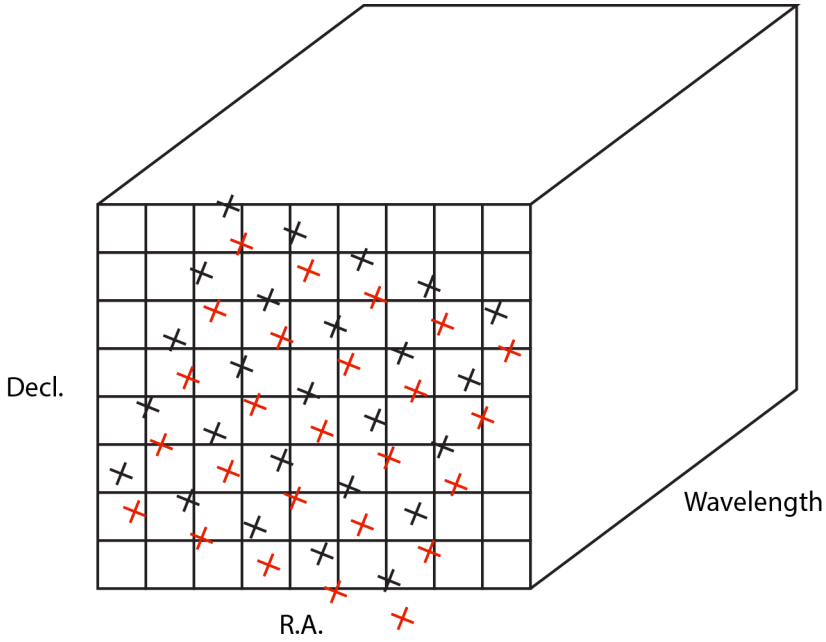


Figure 1: Schematic of two dithered exposures mapped to the IFU output coordinate system (black regular grid). The plus symbols represent the point cloud mapping of detector pixels to effective sampling locations relative to the output coordinate system at a given wavelength. The black points are from 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 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 best algorithm with which to combine the irregularly-distributed samples of the point cloud to a rectilinear data cube is the subject of ongoing study, and depends on both the optical characteristics of the IFU and the science goals of a particular observing program. At present, the default method uses a flux-conserving variant of Shepards method in which the value of a given element 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 will introduce the follow definitions:

- xdistance = distance between point in the cloud and spaxel center in units of arc seconds along the x axis
- ydistance = distance between point in the cloud and spaxel center in units of arc seconds along the y axis
- zdistance = distance between point cloud and spaxel center in the lambda dimension in units of microns along the wavelength axis

These distances are then normalized by the IFU cube sample size for the appropriate axis:

- xnormalized = xdistance/(cube sample size in x dimension [cdelt1])
- ynormalized = ydistance/(cube sample size in y dimension [cdelt2])
- znormalized = zdistance/(cube sample size in z dimension [cdelt3])

The final spaxel 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 spaxel. 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 spaxel, the spaxel flux K = $\frac{\sum_{i=1}^{n} Flux_i w_i}{\sum_{i=1}^{n} w_i}$

where

$w_i = \frac{1.0}{\sqrt{(xnormalized_i^2 + ynormalized_i^2 + znormalized_i^2)^p}}$

The default value for *p* is 2, although the optimal choice for this value (along with the size of the region of influence and the cube sampling scale) is still under study. Similarly, other algorithms such as a 3d generalization of the drizzle algorithm are also being studied and may provide better performance for some science applications.

Additional constraints for weighting=MIRIPSF

For MIRI the weighting function can be adapted to use the width of the PSF and LSF in weighting the point cloud members within the ROI centered on the spaxel. The width of the MIRI PSF varies with wavelength, broader for longer wavelengths. The resolving power of the MRS varies with wavelength and band. Adjacent point-cloud elements may in fact originate from different exposures rotated from one another and even from different spectral bands. In order to properly weight the MIRI data the distances between the point cloud element and spaxel the distances are determined in the alpha-beta coordinate system and then normalized by the width of the PSF and the LSF. To weight in the alpha-beta coordinates system each cube spaxel center must be mapped to the alpha-beta system corresponding to the channel-band of the point cloud member. The xdistance and ydistances are redefined to mean:

- xdistance = distance between point in the cloud and spaxel center along the alpha dimension in units of arc seconds
- ydistance = distance between point in the cloud and spaxel center along the beta dimension in units of arc seconds
- zdistance = distance between point cloud and spaxel center in the lambda dimension in units of microns along the wavelength axis

The spatial distances are then normalized by PSF width and the spectral distance is normalized by the LSF:

- xnormalized = xdistance/(width of the PSF in the alpha dimension in units of arc seconds)
- ynormalized = ydistance/(width of the PSF in the beta dimension in units of arc seconds)
- znormalized = zdistance/(width of LSF in lambda dimension in units of microns)

Step Arguments

As discussed earlier, the input to the `cube_build` step can take many forms, containing data from one or more wavelength bands for each of the MIRI and NIRSpec IFUs. The following step arguments can be used to control

which 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 single-band cubes are selected, 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'`. If this argument is not specified, the output will be a set of IFU cubes, one for each channel/sub-channel combination contained in the input data.

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, so IFU cubes are created either per sub-channel or using all the sub-channels of the data. If this argument is not specified, a set of IFU cubes is created, one for each band. 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 covers.

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` is the default mode and creates IFU cubes containing only one band (channel/sub-channel or grating/filter combination).
- `output_type = channel` combines all the MIRI channels in the data or set by the channel option into a single IFU cube.
- `output_type = grating` combines all the gratings in the NIRSpec data or set by the grating option into a single IFU cube.
- `output_type = multi` combines data into a single "uber" IFU cube. 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.

scale1 The output cube's spaxel size in axis 1 (spatial).

scale2 The output cube's spaxel size in axis 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.

coord_system [string] Options are ra-dec and alpha-beta. The alpha-beta option is a special coordinate system for MIRI data and should only be used by advanced users.

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:

rios [float] The radius of the region of interest in the spatial dimensions.

riow [float] The size of the region of interest in the spectral dimension.

There are two arguments that control how to interpolate the point cloud values:

weighting [string] The type of weighting to use when combining points cloud fluxes to represent the spaxel flux. Allowed values are STANDARD and MIRPSF. This defines how the distances between the point cloud members and spaxel centers are determined. The default value is STANDARD and the distances are determined in the cube output coordinate system. STANDARD is the only option available for NIRSpec. If set to MIRIPSF, the distances are determined in the alpha-beta coordinate system of the point cloud member and are normalized by the PSF and LSF. For more details on how the weight of the point cloud members are used in determining the final spaxel flux see the *Algorithm* section.

weight_power [float] Controls the weighting of the distances between the point cloud member and spaxel center. The weighting function used for determining the spaxel flux was given in the *Algorithm* description: spaxel flux K = $\frac{\sum_{i=1}^{n} Flux_i w_i}{\sum_{i=1}^{n} w_i}$

where n = the number of point cloud points within the region of interest of spaxel flux K

$w_i = 1.0\sqrt{(xnormalized^2 + ynormalized^2 + znormalized^2)}^p$

by default currently p=2, but is controlled by the `weight_power` argument.

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 cube_build.cfg MIRM103-Q0-SHORT_495_cal.fits --ch=1 --band=SHORT
```

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 cube_build.cfg 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"},
```

(continues on next page)

(continued from previous page)

```
                {"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 cube_build.cfg 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 cube_build.cfg 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 cube_build.cfg 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 File

There are two types of reference files used by the cube_build step. The first type holds the default cube parameters used in setting up the output IFU Cube. The reftype for this reference file is *cubepars* and there is a reference file of this type for MIRI data and one for NIRSPEC data. These files contain tables for each band of the spatial and spectral size and the size of the region of interest to use to construct the IFU cube. If more than one band is used to build

the IFU cube, then the final spatial and spectral size will be the smallest one from the list of input bands. Currently cube_build can only produce IFU cubes with a linear spatial and spectral dimension. In the future we plan to allow a varying spectral step with wavelength.

The other type of reference file pertains only to MIRI data and contains the width of the PSF and LSF per band. The reftype for this reference file is *resol*. This information is used if the weight function incorporates the size of the psf and lsf, i.e. –weighting = miripsf

CRDS Selection Criteria

The cube parameter reference file selection is based on Instrument. CRDS selection criteria for the MIRI resolution reference file is also based on Instrument (a N/Q is returned for NIRSPEC data).

Cube Building Parameter Reference File Format

The cube parameter reference files are FITS files with BINTABLE extensions. The FITS primary data array is assumed to be empty. The MIRI cube parameter file contains three BINTABLE extensions, while the NIRSPEC file contains five BINTABLE extensions. In both files the first extension contains the spatial and spectral cube sample size for each band. The second extension holds the Modified Shepard weighting values to use for each band. The third extension will be used in Build 7.2 and contains the wavelengths and associated region of interest size to use if the IFU cubes are created from several bands and the final output is to have an IFU cube of varying spectral scale. In the case of MIRI the twelve spectral bands can be combined into a single IFU cube an all the information to create cubes of varying wavelength sampling is contained in this third BINTABLE extension. However for NIRSPEC data there are three types of multi-band cubes: PRISM, MEDIUM and HIGH resolution. The third, forth and fifth BINTABLE extensions in the NIRSPEC reference file contains the wavelength sampling and region of interest size to use for PRISM, MEDIUM resolution, and HIGH resolution multi-band cubes, respectively.

MIRI Resolution reference file

The MIRI resolution reference file is a FITS file with four BINTABLE extensions. The FITS primary data array is assumed to be empty. The first BINTABLE extension contains the RESOLVING_POWER the information to use for each band. This table has 12 rows and 11 columns, one row of information for each band. The parameters in the 11 columns provide the polynomial coefficients to determine the resolving power for band that row corresponds to. The second BINTABLE extension, PSF_FWHM_ALPHA, has a format of 1 row and 5 columns. The 5 columns hold the polynomial coefficients for determining the alpha PSF size. The third BINTABLE extension, PSF_FWHM_BETA, has a format of 1 row and 5 columns. The 5 columns hold the polynomial coefficients for determining the beta PSF size.

jwst.cube_build Package

Classes

CubeBuildStep([name, parent, config_file, ...])	CubeBuildStep: Creates a 3-D spectral cube from a given association, single model, single input file, or model container.

CubeBuildStep

class jwst.cube_build.**CubeBuildStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

CubeBuildStep: Creates a 3-D spectral cube from a given association, single model, single input file, or model container. Input parameters allow the spectral cube to be built from a provided channel/subchannel (MIRI) or grating/filer (NIRSPEC)

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.
- ****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

reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.
read_user_input()	Short Summary

Attributes Documentation

reference_file_types = ['cubepar', 'resol']

spec = "\n channel = option('1','2','3','4','all',default='all') # Options: 1,2,3,4,

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.

read_user_input()

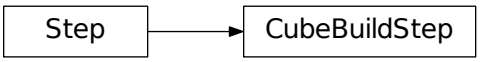
figure out if any of the input paramters channel,band,filter or grating have been set. If they have been check that they are valid and fill in input_pars paramters

Parameters **none** –

Returns

- *self.pars_input['channel']*
- *self.pars_input['sub_channel']*
- *self.pars_input['grating']*
- *self.pars_input['filter']*

Class Inheritance Diagram



12.1.11 Dark Current Subtraction

Description

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 dark current step removes dark current from a JWST exposure by subtracting dark current data stored in a dark reference file.

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 frame-by-frame from the science exposure.

If the science exposure used `nframes>1` or `groupgap>0`, the dark reference file data are reconstructed internally to match the frame averaging and groupgap settings of the science exposure. The reconstructed dark data are constructed by averaging `nframes` adjacent dark frames and skipping `groupgap` intervening frames.

The frame-averaged dark is constructed using the following scheme:

- SCI arrays are computed as the mean of the original dark SCI arrays
- ERR arrays are computed as the uncertainty of the mean, using $\frac{\sqrt{\sum \mathrm{ERR}^2}}{nframes}$

For each integration in the input science exposure, the averaged dark data are then subtracted, group-by-group, from the science exposure groups, as follows:

- Each SCI group of the dark data are subtracted from the corresponding SCI group of the science data
- The ERR arrays of the science data are not modified

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.

The dark DQ array is combined with the science exposure PIXELDQ array using a bitwise OR operation.

Note: If the input science exposure contains more frames 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.

Reference File

The dark current step uses a DARK reference file.

CRDS Selection Criteria

Dark reference files are selected on the basis of INSTRUME, DETECTOR, and SUBARRAY values for the input science data set. For MIRI exposures, the value of READPATT is used as an additional selection criterion.

DARK Reference File Format

Dark 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 extensions for darks used with the Near-IR instruments are as follows:

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

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:

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

The BINTABLE extension in dark reference files contains the bit assignments used in the DQ array. It uses `EXTNAME=DQ_DEF` and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

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 be saved to that file.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

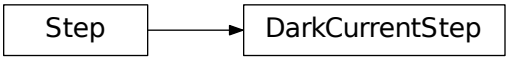
reference_file_types = ['dark']

spec = '\n dark_output = output_file(default = None) # Dark model or averaged dark sub

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



12.1.12 Data Models

About models

The purpose of the data model is to abstract away the peculiarities of the underlying file format. The same data model may be used for data created from scratch in memory, or loaded from FITS or ASDF files or some future file format.

Hierarchy of models

There are different data model classes for different kinds of data.

One model instance, many arrays

Each model instance generally has many arrays that are associated with it. For example, the `ImageModel` class has the following arrays associated with it:

- `data`: The science data

- `dq`: The data quality array
- `err`: The error array

The shape of these arrays must be broadcast-compatible. If you try to assign an array to one of these members that is not broadcast-compatible with the data array, an exception is raised.

Working with models

Creating a data model from scratch

To create a new `ImageModel`, just call its constructor. To create a new model where all of the arrays will have default values, simply provide a shape as the first argument:

```
from jwst.datamodels import ImageModel
with ImageModel((1024, 1024)) as im:
    ...
```

In this form, the memory for the arrays will not be allocated until the arrays are accessed. This is useful if, for example, you don't need a data quality array – the memory for such an array will not be consumed:

```
# Print out the data array.  It is allocated here on first access
# and defaults to being filled with zeros.
print(im.data)
```

If you already have data in a numpy array, you can also create a model using that array by passing it in as a data keyword argument:

```
data = np.empty((50, 50))
dq = np.empty((50, 50))
with ImageModel(data=data, dq=dq) as im:
    ...
```

Creating a data model from a file

The *jwst.datamodels.open* function is a convenient way to create a model from a file on disk. It may be passed any of the following:

- a path to a FITS file
- a path to an ASDF file
- a `astropy.io.fits.HDUList` object
- a readable file-like object

The file will be opened, and based on the nature of the data in the file, the correct data model class will be returned. For example, if the file contains 2-dimensional data, an `ImageModel` instance will be returned. You will generally want to instantiate a model using a `with` statement so that the file will be closed automatically when exiting the `with` block.

```
from jwst import datamodels
with datamodels.open("myimage.fits") as im:
    assert isinstance(im, datamodels.ImageModel)
```

If you know the type of data stored in the file, or you want to ensure that what is being loaded is of a particular type, use the constructor of the desired concrete class. For example, if you want to ensure that the file being opened contains 2-dimensional image data:

```
from jwst.datamodels import ImageModel
with ImageModel("myimage.fits") as im:
    # raises exception if myimage.fits is not an image file
    pass
```

This will raise an exception if the file contains data of the wrong shape.

Saving a data model to a file

Simply call the `save` method on the model instance. The format to save into will either be deduced from the filename (if provided) or the `format` (https://docs.python.org/3/library/functions.html#format) keyword argument:

```
im.save("myimage.fits")
```

Note: Unlike `astropy.io.fits`, `save` always clobbers the output file.

It also accepts a writable file-like object (opened in binary mode). In that case, a format must be specified:

```
with open("myimage.fits", "wb") as fd:
    im.save(fd, format="fits")
```

Copying a model

To create a new model based on another model, simply use its `copy` (https://docs.python.org/3/library/copy.html#module-copy) method. This will perform a deep-copy: that is, no changes to the original model will propagate to the new model:

```
new_model = old_model.copy()
```

It is also possible to copy all of the known metadata from one model into a new one using the update method:

```
new_model.update(old_model)
```

History information

Models contain a list of history records, accessed through the `history` attribute. This is just an ordered list of strings – nothing more sophisticated.

To get to the history:

```
model.history
```

To add an entry to the history:

```
model.history.append("Processed through the frobulator step")
```

These history entries are stored in `HISTORY` keywords when saving to FITS format.

Converting from `astropy.io.fits`

This section describes how to port code that uses `astropy.io.fits` to use *`jwst.datamodels`*.

Opening a file

Instead of:

```
astropy.io.fits.open("myfile.fits")
```

use:

```
from jwst.datamodels import ImageModel
with ImageModel("myfile.fits") as model:
    ...
```

In place of `ImageModel`, use the type of data one expects to find in the file. For example, if spectrographic data is expected, use `SpecModel`. If it doesn't matter (perhaps the application is only sorting FITS files into categories) use the base class `DataModel`.

An alternative is to use:

```
from jwst import datamodels
with datamodels.open("myfile.fits") as model:
    ...
```

The `datamodels.open()` method checks if the `DATAMODL` FITS keyword has been set, which records the DataModel that was used to create the file. If the keyword is not set, then `datamodels.open()` does its best to guess the best DataModel to use.

Accessing data

Data should be accessed through one of the pre-defined data members on the model (`data`, `dq`, `err`). There is no longer a need to hunt through the HDU list to find the data.

Instead of:

```
hdulist['SCI'].data
```

use:

```
model.data
```

Accessing keywords

The data model hides direct access to FITS header keywords. Instead, use the *Metadata* tree.

There is a convenience method, `find_fits_keyword` to find where a FITS keyword is used in the metadata tree:

```
>>> from jwst.datamodels import DataModel
# First, create a model of the desired type
>>> model = DataModel()
>>> model.find_fits_keyword('DATE-OBS')
[u'meta.observation.date']
```

This information shows that instead of:

```
print(hdulist[0].header['DATE-OBS'])
```

use:

```
print(model.meta.observation.date)
```

Extra FITS keywords

When loading arbitrary FITS files, there may be keywords that are not listed in the schema for that data model. These "extra" FITS keywords are put under the model in the `_extra_fits` namespace.

Under the `_extra_fits` namespace is a section for each header data unit, and under those are the extra FITS keywords. For example, if the FITS file contains a keyword `FOO` in the primary header, its value can be obtained using:

```
model._extra_fits.PRIMARY.FOO
```

This feature is useful to retain any extra keywords from input files to output products.

To get a list of everything in `_extra_fits`:

```
model._extra_fits._instance
```

returns a dictionary of of the instance at the model._extra_fits node.

`_instance` can be used at any node in the tree to return a dictionary of rest of the tree structure at that node.

Data model attributes

The purpose of the data model is to abstract away the peculiarities of the underlying file format. The same data model may be used for data created from scratch in memory, loaded from FITS or ASDF files, or from some other future format.

Calling sequences of models

List of current models

The current models are as follows:

`AmiLgModel`, `AsnModel`, `BarshadowModel`, `CameraModel`, `CollimatorModel`, `ContrastModel`, `CubeModel`, `IFUCubeModel`, `DarkModel`, `DarkMIRIModel`, `DisperserModel`, `DistortionModel`, `DistortionMRSModel`, `DrizParsModel`, `DrizProductModel`, `Extract1dImageModel`, `FilteroffsetModel`, `FlatModel`, `CubeFlatModel`, `NRSFlatModel`, `NirspecFlatModel`, `NirspecQuadFlatModel`, `FOREModel`, `FPAModel`, `FringeModel`, `GainModel`, `GLS_RampFitModel`, `NIRCAMGrismModel`, `NIRISSGrismModel`, `GuiderCalModel`, `GuiderRawModel`, `ImageModel`, `IFUImageModel`, `IFUCubeParsModel`, `NirspecIFUCubeParsModel`, `MiriIFUCubeParsModel`, `IFUFOREModel`, `IFUPostModel`, `IFUSlicerModel`, `IPCModel`, `IRS2Model`, `LastFrameModel`, `Level1bModel`, `LinearityModel`, `MaskModel`, `MSAModel`, `ModelContainer`, `MultiExposureModel`, `MultiProductModel`, `MultiSlitModel`, `MultiSpecModel`, `OTEModel`, `OutlierParsModel`, `PathlossModel`, `PersistenceSatModel`, `PhotomModel`, `FgsPhotomModel`, `MiriImgPhotomModel`,

MiriMrsPhotomModel, NircamPhotomModel, NirissPhotomModel, NirspecPhotomModel, NirspecFSPhotomModel, PixelAreaModel, PsfMaskModel, QuadModel, RampModel, MIRIRampModel, RampFitOutputModel, ReadnoiseModel, ReferenceFileModel, ReferenceImageModel, ReferenceCubeModel, ReferenceQuadModel, RegionsModel, ResetModel, ResolutionModel, MiriResolutionModel, RSCDModel, SaturationModel, SpecModel, SpecwcsModel, StrayLightModel, SuperBiasModel, ThroughputModel, TrapDensityModel, TrapParsModel, TrapsFilledModel, TsoPhotModel, WaveCorrModel, WavelengthrangeModel, WfssBkgModel

Commonly used attributes

Here are a few model attributes that are used by some of the pipeline steps.

For uncalibrated data `_uncal.fits`. Getting the number of integrations and the number of groups from the first and second axes assumes that the input data array is 4-D data. Pixel coordinates in the data extensions are 1-indexed as in FORTRAN and FITS headers, not 0-indexed as in Python.

- `input_model.data.shape[0]`: number of integrations
- `input_model.data.shape[1]`: number of groups
- `input_model.meta.exposure.nframes`: number of frames per group
- **`input_model.meta.exposure.groupgap`: number of frames dropped** between groups
- `input_model.meta.subarray.xstart`: starting pixel in X (1-based)
- `input_model.meta.subarray.ystart`: starting pixel in Y (1-based)
- `input_model.meta.subarray.xsize`: number of columns
- `input_model.meta.subarray.ysize`: number of rows

The `data`, `err`, `dq`, etc., attributes of most models are assumed to be numpy.ndarray arrays, or at least objects that have some of the attributes of these arrays. numpy is used explicitly to create these arrays in some cases (e.g. when a default value is needed). The `data` and `err` arrays are a floating point type, and the data quality arrays are an integer type.

Some of the step code makes assumptions about image array sizes. For example, full-frame MIRI data have 1032 columns and 1024 rows, and all other detectors have 2048 columns and rows; anything smaller must be a subarray. Also, full-frame MIRI data are assumed to have four columns of reference pixels on the left and right sides (the reference output array is stored in a separate image extension). Full-frame data for all other instruments have four columns or rows of reference pixels on each edge of the image.

DataModel Base Class

class jwst.datamodels.**DataModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Base class of all of the data models.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape

  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

add_schema_entry(*position*, *new_schema*)
Extend the model's schema by placing the given new_schema at the given dot-separated position in the tree.

Parameters

- **position** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) –
- **new_schema** (*schema tree*) –

copy(*memo=None*)
Returns a deep copy of this model.

extend_schema(*new_schema*)
Extend the model's schema using the given schema, by combining it in an "allOf" array.

Parameters **new_schema** (*schema tree*) –

find_fits_keyword(*keyword*, *return_result=True*)
Utility function to find a reference to a FITS keyword in this model's schema. This is intended for interactive use, and not for use within library code.

Parameters **keyword** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A FITS keyword name

Returns **locations** – If `return_result` is `True` (https://docs.python.org/3/library/constants.html#True), a list of the locations in the schema where this FITS keyword is used. Each element is a dot-separated path.

Return type list of str

Example

```
>>> model.find_fits_keyword('DATE-OBS')
['observation.date']
```

classmethod from_asdf(*init*, *schema=None*)
Load a data model from a ASDF file.

Parameters

- **init** (*file path, file object, asdf.AsdfFile object*) –
  - file path: Initialize from the given file
  - readable file object: Initialize from the given file object
  - asdf.AsdfFile: Initialize from the given `AsdfFile`.
- **schema** – Same as for `__init__`

Returns **model**

Return type DataModel instance

classmethod from_fits(*init*, *schema=None*)
Load a model from a FITS file.

Parameters

- **init** (*file path, file object, astropy.io.fits.HDUList*) –
  - file path: Initialize from the given file
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given `HDUList`.
- **schema** – Same as for `__init__`

Returns **model**

Return type DataModel instance

get_fits_wcs(*hdu_name='SCI'*, *hdu_ver=1*, *key=' '*)
Get a `astropy.wcs.WCS` object created from the FITS WCS information in the model.

Note that modifying the returned WCS object will not modify the data in this model. To update the model, use *set_fits_wcs*.

Parameters

- **hdu_name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the HDU to get the WCS from. This must use named HDU's, not numerical order HDUs. To get the primary HDU, pass `'PRIMARY'`.
- **key** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of a particular WCS transform to use. This may be either `' '` or `'A'-'Z'` and corresponds to the `"a"` part of the `CTYPEia` cards. *key* may only be provided if *header* is also provided.
- **hdu_ver** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – The extension version. Used when there is more than one extension with the same name. The default value, 1, is the first.

Returns **wcs** – The type will depend on what libraries are installed on this system.

Return type `astropy.wcs.WCS` or `pywcs.WCS` object

get_item_as_json_value(*key*)
Equivalent to __getitem__, except returns the value as a JSON basic type, rather than an arbitrary Python type.

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

history
Get the history as a list of entries

info()
Return datatype and dimension for each array or table

items()
Iterates over all of the schema items in a flat way.

Each element is a pair (`key`, `value`). Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as:

```
("meta.observation.date": "2012-04-22T03:22:05.432")
```

iteritems()
Iterates over all of the schema items in a flat way.

Each element is a pair (`key`, `value`). Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as:

```
("meta.observation.date": "2012-04-22T03:22:05.432")
```

iterkeys()
Iterates over all of the schema keys in a flat way.

Each result of the iterator is a `key`. Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as the string `"meta.observation.date"`.

itervalues()
Iterates over all of the schema values in a flat way.

keys()
Iterates over all of the schema keys in a flat way.

Each result of the iterator is a `key`. Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as the string `"meta.observation.date"`.

my_attribute(*attr*)
Test if attribute is part of the NDData interface

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

read(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

save (*path*, *dir_path=None*, **args*, ***kwargs*)

Save to either a FITS or ASDF file, depending on the path.

Parameters

- **path** (*string or func*) – File path to save to. If function, it takes one argument with is model.meta.filename and returns the full path string.
- **dir_path** (*string*) – Directory to save to. If not None, this will override any directory information in the `path`

Returns **output_path** – The file path the model was saved in.

Return type str (https://docs.python.org/3/library/stdtypes.html#str)

search_schema (*substring*)

Utility function to search the metadata schema for a particular phrase.

This is intended for interactive use, and not for use within library code.

The searching is case insensitive.

Parameters **substring** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The substring to search for.

Returns locations

Return type list of tuples

set_fits_wcs(*wcs*, *hdu_name='SCI'*)
Sets the FITS WCS information on the model using the given `astropy.wcs.WCS` object.

Note that the "key" of the WCS is stored in the WCS object itself, so it can not be set as a parameter to this method.

Parameters

- **wcs** (`astropy.wcs.WCS` or `pywcs.WCS` object) – The object containing FITS WCS information
- **hdu_name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the HDU to set the WCS from. This must use named HDU's, not numerical order HDUs. To set the primary HDU, pass `'PRIMARY'`.

to_asdf(*init*, **args*, ***kwargs*)
Write a DataModel to an ASDF file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `asdf.AsdfFile.write_to`.

to_fits(*init*, **args*, ***kwargs*)
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

to_flat_dict(*include_arrays=True*)
Returns a dictionary of all of the schema items as a flat dictionary.

Each dictionary key is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the dictionary as:

```
{ "meta.observation.date": "2012-04-22T03:22:05.432" }
```

update(*d*, *only=''*)
Updates this model with the metadata elements from another model.

Parameters

- **d** (*model or dictionary-like object*) – The model to copy the metadata elements from. Can also be a dictionary or dictionary of dictionaries or lists.
- **only** (*only update the named hdu from extra_fits, e.g.*) – only='PRIMARY'. Can either be a list of hdu names or a single string. If left blank, update all the hdus.

validate()
Re-validate the model instance againsst its schema

validate_required_fields()
Walk the schema and make sure all required fields are in the model

values()
Iterates over all of the schema values in a flat way.

Specific Model Classes

class jwst.datamodels.**AmiLgModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
A data model for AMI LG analysis results.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

class jwst.datamodels.**AsnModel**(*init=None*, ***kwargs*)
A data model for association tables.

class jwst.datamodels.**BarshadowModel**(*init=None*, ***kwargs*)
A data model for Bar Shadow correction information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – Array defining the bar shadow correction as a function of Y and wavelength.
- **variance** (*numpy array*) – Variance array.

class jwst.datamodels.**CameraModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
A model for a reference file of type "camera".

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

class jwst.datamodels.**CollimatorModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
A model for a reference file of type "collimator".

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

class jwst.datamodels.**ContrastModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
A data model for coronagraphic contrast curve files.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –

- **==== ===== ============** (=========) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

class jwst.datamodels.**CubeModel**(*init=None*, ***kwargs*)
A data model for 3D image cubes.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 3-D.
- **dq** (*numpy array*) – The data quality array. 3-D.
- **err** (*numpy array*) – The error array. 3-D
- **zeroframe** (*numpy array*) – The zero-frame array. 3-D
- **relsens** (*numpy array*) – The relative sensitivity array.
- **int_times** (*table*) – The int_times table
- **area** (*numpy array*) – The pixel area array. 2-D
- **wavelength** (*numpy array*) – The wavelength array. 2-D
- **var_poisson** (*numpy array*) – The variance due to Poisson noise array. 3-D
- **var_rnoise** (*numpy array*) – The variance due to read noise array. 3-D

class jwst.datamodels.**IFUCubeModel**(*init=None*, ***kwargs*)
A data model for 3D IFU cubes.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 3-D.
- **dq** (*numpy array*) – The data quality array. 3-D.
- **err** (*numpy array*) – The error array. 3-D
- **weightmap** (*numpy array*) – The weight map array. 3-D
- **wavetable** (*1-D table*) – Optional table of wavelengths of IFUCube slices

class jwst.datamodels.**DarkModel**(*init=None*, ***kwargs*)
A data model for dark reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**DarkMIRIModel**(*init=None*, ***kwargs*)
A data model for dark MIRI reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data (integration dependent)
- **dq** (*numpy array*) – The data quality array. (integration dependent)
- **err** (*numpy array (integration dependent)*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**DisperserModel**(*init=None*, *angle=None*, *gwa_tiltx=None*, *gwa_tilty=None*, *kcoef=None*, *lcoef=None*, *tcoef=None*, *pref=None*, *tref=None*, *theta_x=None*, *theta_y=None*, *theta_z=None*, *groovedensity=None*, ***kwargs*)

A model for a NIRSPEC reference file of type "disperser".

on_save(*path=None*)

This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()

Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()

Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**DistortionModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)

A model for a reference file of type "distortion".

validate()

Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**DistortionMRSModel**(*init=None*, *x_model=None*, *y_model=None*, *alpha_model=None*, *beta_model=None*, *bzero=None*, *bdel=None*, *input_units=None*, *output_units=None*, ***kwargs*)

A model for a reference file of type "distortion" for the MIRI MRS.

on_save(*path=None*)

This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**DrizParsModel**(*init=None*, ***kwargs*)
A data model for drizzle parameters reference tables.

class jwst.datamodels.**DrizProductModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
A data model for drizzle-generated products.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

class jwst.datamodels.**Extract1dImageModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

A data model for the extract_1d reference image array.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – An array of values that define the extraction regions.
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

class jwst.datamodels.**FilteroffsetModel**(*init=None*, *filters=None*, ***kwargs*)

A model for a NIRSPEC reference file of type "disperser".

on_save(*path=None*)

This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**FlatModel**(*init=None*, ***kwargs*)
A data model for 2D flat-field images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D.
- **dq** (*numpy array*) – The data quality array. 2-D.
- **err** (*numpy array*) – The error array. 2-D.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**NRSFlatModel**(*init=None*, ***kwargs*)
A base class for NIRSpec flat-field reference file models.

class jwst.datamodels.**NirspecFlatModel**(*init=None*, ***kwargs*)
A data model for NIRSpec flat-field reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D or 3-D.
- **dq** (*numpy array*) – The data quality array. 2-D or 3-D.
- **err** (*numpy array*) – The error array. 2-D or 3-D.
- **wavelength** (*numpy array*) – The wavelength for each plane of the `data` array. This will only be needed if `data` is 3-D.
- **flat_table** (*numpy array*) – A table of wavelengths and flat-field values, to specify the component of the flat field that can vary over a relatively short distance (can be pixel-to-pixel).

class jwst.datamodels.**NirspecQuadFlatModel**(*init=None*, ***kwargs*)
A data model for NIRSpec flat-field files that differ by quadrant.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D or 3-D.
- **dq** (*numpy array*) – The data quality array. 2-D or 3-D.
- **err** (*numpy array*) – The error array. 2-D or 3-D.
- **wavelength** (*numpy array*) – The wavelength for each plane of the `data` array. This will only be needed if `data` is 3-D.
- **flat_table** (*numpy array*) – A table of wavelengths and flat-field values, to specify the component of the flat field that can vary over a relatively short distance (can be pixel-to-pixel).
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**FOREModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
A model for a reference file of type "fore".

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**FPAModel**(*init=None*, *nrs1_model=None*, *nrs2_model=None*, ***kwargs*)
A model for a NIRSPEC reference file of type "fpa".

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**FringeModel**(*init=None*, ***kwargs*)
A data model for 2D fringe correction images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**GainModel**(*init=None*, ***kwargs*)
A data model for 2D gain.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.

- **data** (*numpy array*) – The 2-D gain array

class jwst.datamodels.**GLS_RampFitModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

A data model for the optional output of the ramp fitting step for the GLS algorithm.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

class jwst.datamodels.**NIRCAMGrismModel**(*init=None*, *displ=None*, *dispx=None*, *dispy=None*, *invdispl=None*, *invdispx=None*, *invdispy=None*, *orders=None*, ***kwargs*)

A model for a reference file of type "specwcs" for NIRCAM grisms.

This reference file contains the models for wave, x, and y polynomial solutions that describe dispersion through the grism

to_fits()

Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**NIRISSGrismModel**(*init=None*, *displ=None*, *dispx=None*, *dispy=None*, *invdispl=None*, *orders=None*, *fwcpos_ref=None*, ***kwargs*)
A model for a reference file of type "specwcs" for NIRISS grisms.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**GuiderCalModel**(*init=None*, ***kwargs*)
A data model for FGS pipeline output files

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 3-D
- **dq** (*numpy array*) – The data quality array. 2-D
- **err** (*numpy array*) – The error array. 3-D
- **plan_star_table** (*table*) – The planned reference star table
- **flight_star_table** (*table*) – The flight reference star table
- **pointing_table** (*table*) – The pointing table
- **centroid_table** (*table*) – The centroid packet table
- **track_sub_table** (*table*) – The track subarray table

class jwst.datamodels.**GuiderRawModel**(*init=None*, ***kwargs*)
A data model for FGS pipeline input files

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 4-D
- **dq** (*numpy array*) – The data quality array. 2-D.
- **err** (*numpy array*) – The error array. 4-D.
- **plan_star_table** (*table*) – The planned reference star table
- **flight_star_table** (*table*) – The flight reference star table
- **pointing_table** (*table*) – The pointing table
- **centroid_table** (*table*) – The centroid packet table
- **track_sub_table** (*table*) – The track subarray table

class jwst.datamodels.**ImageModel**(*init=None*, ***kwargs*)
A data model for 2D images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **kwargs** (*numpy array*) – The name and value of any array mentioned in the schema to be initialized through the function call.

class jwst.datamodels.**IFUImageModel**(*init=None*, ***kwargs*)
A data model for 2D IFU images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **relsens2d** (*numpy array*) – The relative sensitivity 2D array.

class jwst.datamodels.**IFUCubeParsModel**(*init=None*, ***kwargs*)
A data model for IFU Cube parameters reference tables.

class jwst.datamodels.**NirspecIFUCubeParsModel**(*init=None*, ***kwargs*)
A data model for Nirspec ifucubepars reference files.

class jwst.datamodels.**MiriIFUCubeParsModel**(*init=None*, ***kwargs*)
A data model for MIRI mrs ifucubepars reference files.

class jwst.datamodels.**IFUFOREModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
A model for a NIRSPEC reference file of type "ifufore".

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

class jwst.datamodels.**IFUPostModel**(*init=None*, *slice_models=None*, ***kwargs*)
A model for a NIRSPEC reference file of type "ifupost".

Parameters

- **init** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A file name.
- **slice_models** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – A dictionary with slice transforms with the following entries: {"slice_N": {'linear': astropy.modeling.Model,

  'xpoly': astropy.modeling.Model, 'xpoly_distortion': astropy.modeling.Model, 'ypoly': astropy.modeling.Model, 'ypoly_distortion': astropy.modeling.Model, }

  }

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**IFUSlicerModel**(*init=None*, *model=None*, *data=None*, ***kwargs*)
A model for a NIRSPEC reference file of type "ifuslicer".

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**IPCModel**(*init=None*, ***kwargs*)
A data model for IPC kernel checking information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The deconvolution kernel (a very small image).

class jwst.datamodels.**IRS2Model**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
A data model for the IRS2 refpix reference file.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **irs2_table** (*numpy array*) – A table with 8 columns and 2916352 (2048 * 712 * 2) rows. All values are float, but these are interpreted as alternating real and imaginary parts

(real, imag, real, imag, ...) of complex values. There are four columns for ALPHA and four for BETA.

- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

class jwst.datamodels.**LastFrameModel**(*init=None*, ***kwargs*)

A data model for Last frame correction reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**Level1bModel**(*init=None*, ***kwargs*)

A data model for raw 4D ramps level-1b products.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data
- **zeroframe** (*numpy array*) – The zero-frame data
- **refout** (*numpy array*) – The MIRI reference output data

- **group** (*table*) – The group parameters table
- **int_times** (*table*) – The int_times table

class jwst.datamodels.**LinearityModel**(*init=None*, ***kwargs*)
A data model for linearity correction information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **coeffs** (*numpy array*) – Coefficients defining the nonlinearity function.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

class jwst.datamodels.**MaskModel**(*init=None*, ***kwargs*)
A data model for 2D masks.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

class jwst.datamodels.**MSAModel**(*init=None*, *models=None*, *data=None*, ***kwargs*)
A model for a NIRSPEC reference file of type "msa".

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**ModelContainer**(*init=None*, *persist=True*, ***kwargs*)
A container for holding DataModels.

This functions like a list for holding DataModel objects. It can be iterated through like a list, DataModels within the container can be addressed by index, and the datamodels can be grouped into a list of lists for grouped looping, useful for NIRCam where grouping together all detectors of a given exposure is useful for some pipeline steps.

Parameters

- **init** (*file path, list of DataModels, or None* (https://docs.python.org/3/library/constants.html#None)) –
  - file path: initialize from an association table
  - list: a list of DataModels of any type
  - None: initializes an empty *ModelContainer* instance, to which DataModels can be added via the `append()` method.
- **persist** (*boolean. If True, do not close model after opening it*) –

Examples

```
>>> container = datamodels.ModelContainer('example_asn.json')
>>> for dm in container:
...     print(dm.meta.filename)
```

Say the association was a NIRCam dithered dataset. The *models_grouped* attribute is a list of lists, the first index giving the list of exposure groups, with the second giving the individual datamodels representing each detector in the exposure (2 or 8 in the case of NIRCam).

```
>>> total_exposure_time = 0.0
>>> for group in container.models_grouped:
...     total_exposure_time += group[0].meta.exposure.exposure_time
```

```
>>> c = datamodels.ModelContainer()
>>> m = datamodels.open('myfile.fits')
>>> c.append(m)
```

copy (*memo=None*)
Returns a deep copy of the models in this model container.

from_asn (*filepath*, ***kwargs*)
Load fits files from a JWST association file.

Parameters **filepath** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to an association file.

get_recursively (*field*)
Returns a list of values of the specified field from meta.

group_names
Return list of names for the DataModel groups by exposure.

models_grouped
Returns a list of a list of datamodels grouped by exposure.

Data from different detectors of the same exposure will have the same group id, which allows grouping by exposure. The following metadata is used for grouping:

meta.observation.program_number meta.observation.observation_number meta.observation.visit_number meta.observation.visit_group meta.observation.sequence_id meta.observation.activity_id meta.observation.exposure_number

save(*path=None*, *dir_path=None*, *save_model_func=None*, **args*, ***kwargs*)
Write out models in container to FITS or ASDF.

Parameters

- **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or func or None* (https://docs.python.org/3/library/constants.html#None)) –
  - If None, the `meta.filename` is used for each model.
  - If a string, the string is used as a root and an index is appended.
  - If a function, the function takes the two arguments: the value of model.meta.filename and the `idx` index, returning constructed file name.
- **dir_path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Directory to write out files. Defaults to current working dir. If directory does not exist, it creates it. Filenames are pulled from `meta.filename` of each datamodel in the container.
- **save_model_func** (*func or None* (https://docs.python.org/3/library/constants.html#None)) – Alternate function to save each model instead of the models *save* method. Takes one argument, the model, and keyword argument `idx` for an index.

Returns **output_paths** – List of output file paths of where the models were saved.

Return type [str (https://docs.python.org/3/library/stdtypes.html#str)[, ..]]

class jwst.datamodels.**MultiExposureModel**(*init=None*, ***kwargs*)
A data model for multi-slit images derived from numerous exposures. The intent is that all slits in this model are of the same source, with each slit representing a separate exposure of that source.

This model has a special member `exposures` that can be used to deal with an entire slit at a time. It behaves like a list:

```
>>> multislit_model.exposures.append(image_model)
>>> multislit_model.exposures[0]
<ImageModel>
```

Also, there is an extra attribute, `meta`. This will contain the meta attribute from the exposure from which each slit has been taken.

See the module `exp_to_source` for the initial creation of these models. This is part of the Level 3 processing of multi-objection observations.

class jwst.datamodels.**MultiProductModel**(*init=None*, ***kwargs*)
A data model for multi-DrizProduct images.

This model has a special member `products` that can be used to deal with each DrizProduct at a time. It behaves like a list:

```
>>> multiprod_model.products.append(image_model)
>>> multislit_model.products[0]
<DrizProductModel>
```

If `init` is a file name or an *DrizProductModel* instance, an empty *DrizProductModel* will be created and assigned to attribute `products[0]`, and the `data`, `wht`, `con`, and `relsens` attributes from the input file or *DrizProductModel* will be copied to the first element of `products`.

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

class jwst.datamodels.**MultiSlitModel**(*init=None*, ***kwargs*)
A data model for multi-slit images.

This model has a special member `slits` that can be used to deal with an entire slit at a time. It behaves like a list:

```
>>> multislit_model.slits.append(image_model)
>>> multislit_model.slits[0]
>>> multislit[0]
<SlitModel>
```

If `init` is a file name or an `ImageModel` or a `SlitModel``instance, an empty ``SlitModel` will be created and assigned to attribute `slits[0]`, and the `data`, `dq`, `err`, `var_rnoise`, `var_poisson``and ``relsens` attributes from the input file or model will be copied to the first element of `slits`.

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

class jwst.datamodels.**MultiSpecModel**(*init=None*, ***kwargs*)
A data model for multi-spec images.

This model has a special member `spec` that can be used to deal with an entire spectrum at a time. It behaves like a list:

```
>>> multispec_model.spec.append(spec_model)
>>> multispec_model.spec[0]
<SpecModel>
```

If `init` is a *SpecModel* instance, an empty *SpecModel* will be created and assigned to attribute `spec[0]`, and the `spec_table` attribute from the input *SpecModel* instance will be copied to the first element of `spec`. *SpecModel* objects can be appended to the `spec` attribute by using its `append` method.

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

Examples

```
>>> output_model = datamodels.MultiSpecModel()
>>> spec = datamodels.SpecModel()        # for the default data type
>>> for slit in input_model.slits:
>>>     slitname = slit.name
>>>     slitmodel = ExtractModel()
>>>     slitmodel.fromJSONFile(extref, slitname)
>>>     column, wavelength, countrate = slitmodel.extract(slit.data)
>>>     otab = np.array(zip(column, wavelength, countrate),
>>>                     dtype=spec.spec_table.dtype)
>>>     spec = datamodels.SpecModel(spec_table=otab)
>>>     output_model.spec.append(spec)
```

class jwst.datamodels.**OTEModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
A model for a reference file of type "ote".

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

class jwst.datamodels.**OutlierParsModel**(*init=None*, ***kwargs*)
A data model for outlier detection parameters reference tables.

class jwst.datamodels.**PathlossModel**(*init=None*, ***kwargs*)
A data model for pathloss correction information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **pointsource** (*numpy array*) – Array defining the pathloss parameter for point sources.
- **psvar** (*numpy array*) – Variance array.
- **uniform** (*numpy array*) – Pathloss parameter for uniform illumination

class jwst.datamodels.**PersistenceSatModel**(*init=None*, ***kwargs*)
A data model for the persistence saturation value (full well).

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**PhotomModel**(*init=None*, ***kwargs*)
A base class for photometric reference file models.

class jwst.datamodels.**FgsPhotomModel**(*init=None*, ***kwargs*)
A data model for FGS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[5000]
  - relresponse: float32[5000]

class jwst.datamodels.**MiriImgPhotomModel**(*init=None*, ***kwargs*)
A data model for MIRI imaging photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - subarray: str[15]
  - photmjsr: float32

- uncertainty: float32
- nelem: int16
- wavelength: float32[500]
- relresponse: float32[500]

class jwst.datamodels.**MiriMrsPhotomModel**(*init=None*, ***kwargs*)
A data model for MIRI MRS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – An array-like object containing the pixel-by-pixel conversion values in units of DN / sec / mJy / pixel.
- **err** (*numpy array*) – An array-like object containing the uncertainties in the conversion values, in the same units as the data array.
- **dq** (*numpy array*) – An array-like object containing bit-encoded data quality flags, indicating problem conditions for values in the data array.
- **dq_def** (*numpy array*) – A table-like object containing the data quality definitions table.
- **pixsiz** (*numpy array*) – An array-like object containing pixel-by-pixel size values, in units of square arcseconds (arcsec^2).

class jwst.datamodels.**NircamPhotomModel**(*init=None*, ***kwargs*)
A data model for NIRCam photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - pupil: str[12]
  - order: int16
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[3000]
  - relresponse: float32[3000]

class jwst.datamodels.**NirissPhotomModel**(*init=None*, ***kwargs*)
A data model for NIRISS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.

- filter: str[12]
- pupil: str[12]
- order: int16
- photmjsr: float32
- uncertainty: float32
- nelem: int16
- wavelength: float32[5000]
- relresponse: float32[5000]

class jwst.datamodels.**NirspecPhotomModel**(*init=None*, ***kwargs*)

A data model for NIRSpec imaging, IFU, and MOS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - grating: str[12]
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[150]
  - relresponse: float32[150]
  - reluncertainty: float32[150]

class jwst.datamodels.**NirspecFSPhotomModel**(*init=None*, ***kwargs*)

A data model for NIRSpec Fixed-Slit (FS) photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - grating: str[12]
  - slit: str[12]
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[150]
  - relresponse: float32[150]

– reluncertainty: float32[150]

class jwst.datamodels.**PixelAreaModel**(*init=None*, ***kwargs*)
A data model for the pixel area map

class jwst.datamodels.**PsfMaskModel**(*init=None*, ***kwargs*)
A data model for coronagraphic 2D PSF mask reference files

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The 2-D mask array

class jwst.datamodels.**QuadModel**(*init=None*, ***kwargs*)
A data model for 4D image arrays.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 4-D.
- **dq** (*numpy array*) – The data quality array. 4-D.
- **err** (*numpy array*) – The error array. 4-D

class jwst.datamodels.**RampModel**(*init=None*, ***kwargs*)
A data model for 4D ramps.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **pixeldq** (*numpy array*) – 2-D data quality array.
- **groupdq** (*numpy array*) – 3-D or 4-D data quality array.
- **err** (*numpy array*) – The error array.
- **group** (*table*) – The group parameters table
- **int_times** (*table*) – The int_times table

class jwst.datamodels.**MIRIRampModel**(*init=None*, ***kwargs*)
A data model for MIRI ramps. Includes the `refout` array.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **pixeldq** (*numpy array*) – 2-D data quality array.
- **groupdq** (*numpy array*) – 3-D or 4-D data quality array.
- **err** (*numpy array*) – The error array.
- **refout** (*numpy array*) – The array of reference output data.
- **group** (*table*) – The group parameters table.

class jwst.datamodels.**RampFitOutputModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
A data model for the optional output of the ramp fitting step.

In the parameter definitions below, n_int is the number of integrations, max_seg is the maximum number of segments that were fit, nreads is the number of reads in an integration, and ny and nx are the height and width of the image.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **slope** (*numpy array (n_int, max_seg, ny, nx)*) –
- **sigslope** (*numpy array (n_int, max_seg, ny, nx)*) –
- **var_poisson** (*numpy array (n_int, max_seg, ny, nx)*) –
- **var_rnoise** (*numpy array (n_int, max_seg, ny, nx)*) –
- **yint** (*numpy array (n_int, max_seg, ny, nx)*) –
- **sigyint** (*numpy array (n_int, max_seg, ny, nx)*) –
- **pedestal** (*numpy array (n_int, max_seg, ny, nx)*) –
- **weights** (*numpy array (n_int, max_seg, ny, nx)*) –
- **crmag** (*numpy array (n_int, max_seg, ny, nx)*) –
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

class jwst.datamodels.**ReadnoiseModel**(*init=None*, ***kwargs*)
A data model for 2D readnoise.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – Read noise for all pixels. 2-D.

class jwst.datamodels.**ReferenceFileModel**(*init=None*, ***kwargs*)
A data model for reference tables

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**ReferenceImageModel**(*init=None*, ***kwargs*)
A data model for 2D reference images

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.

class jwst.datamodels.**ReferenceCubeModel**(*init=None*, ***kwargs*)
A data model for 3D reference images

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.

class jwst.datamodels.**ReferenceQuadModel**(*init=None*, ***kwargs*)
A data model for 4D reference images

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.

class jwst.datamodels.**RegionsModel**(*init=None*, *regions=None*, ***kwargs*)
A model for a reference file of type "regions".

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**ResetModel**(*init=None*, ***kwargs*)
A data model for reset correction reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**ResolutionModel**(*init=None*, ***kwargs*)
A data model for Spectral Resolution parameters reference tables.

class jwst.datamodels.**MiriResolutionModel**(*init=None*, ***kwargs*)
A data model for MIRI Resolution reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by '~jwst.datamodels.DataModel'
- **resolving_power_table** (*table*) – A table containing resolving power of the MRS. THe table consist of 11 columns and 12 rows. Each row corresponds to a band. The columns give the name of band, central wavelength, and polynomial coefficeints (a,b,c) needed to obtain the limits and average value of the spectral resolution.
- **psf_fwhm_alpha_table** (*table*) – A table with 5 columns. Column 1 gives the cutoff wavelength where the polynomials describing alpha FWHM change. Columns 2 and 3 give the polynomial cofficients (a,b) describing alpha FWHM for wavelengths shorter than cuttoff. Columns 4 and 5 give the polynomial coefficients (a,b) describing alpha FWHM for wavelengths longer than the cutoff.
- **psf_fwhm_beta_table** (*table*) – A table with 5 columns. Column 1 gives the cutoff wavelength where the polynomials describing alpha FWHM change. Columns 2 and 3 give the polynomial cofficients (a,b) describing beta FWHM for wavelengths shorter than cuttoff. Columns 4 and 5 give the polynomial coefficients (a,b) describing beta FWHM for wavelengths longer than the cutoff.

class jwst.datamodels.**RSCDModel**(*init=None*, ***kwargs*)
A data model for the RSCD reference file.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.

- **rscd_table** (*numpy array*) – A table with seven columns, three string-valued that identify which row to select, and four float columns containing coefficients.

class jwst.datamodels.**SaturationModel**(*init=None*, ***kwargs*)
A data model for saturation checking information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**SpecModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
A data model for 1D spectra.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **spec_table** (*numpy array*) – A table with at least four columns: wavelength, flux, an error estimate for the flux, and data quality flags.
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –

- **==== ===== ============** –

class jwst.datamodels.**SpecwcsModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)

A model for a reference file of type "specwcs".

validate()

Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**StrayLightModel**(*init=None*, ***kwargs*)

A data model for 2D straylight mask.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – 2-D straylight mask array.

class jwst.datamodels.**SuperBiasModel**(*init=None*, ***kwargs*)

A data model for 2D super-bias images.

class jwst.datamodels.**ThroughputModel**(*init=None*, ***kwargs*)

A data model for filter throughput.

class jwst.datamodels.**TrapDensityModel**(*init=None*, ***kwargs*)

A data model for the trap density of a detector, for persistence.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

class jwst.datamodels.**TrapParsModel**(*init=None*, ***kwargs*)

A data model for trap capture and decay parameters.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **trappars_table** (*numpy array*) – A table with three columns for trap-capture parameters and one column for the trap-decay parameter. Each row of the table is for a different trap family.

class jwst.datamodels.**TrapsFilledModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

A data model for the number of traps filled for a detector, for persistence.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The map of the number of traps filled over the detector, with one plane for each "trap family."
- **init** –

- None: A default data model with no shape
- shape tuple: Initialize with empty data of the given shape
- file path: Initialize from the given file (FITS or ASDF)
- readable file object: Initialize from the given file object
- `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
- A numpy array: Used to initialize the data array
- dict: The object model tree for the data model

schema (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.

extensions (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.

pass_invalid_values (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None

strict_validation (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.

available built-in formats are (*The*) –

==== ===== ============= (*=========*) – Format Read Write Auto-identify

==== ===== ============= –

Yes Yes Yes (*datamodel*) –

==== ===== ============= –

class jwst.datamodels.**TsoPhotModel**(*init=None*, *radii=None*, ***kwargs*)

A model for a reference file of type "tsophot".

on_save(*path=None*)

This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()

Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()

Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**WaveCorrModel**(*init=None*, *apertures=None*, ***kwargs*)

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**WavelengthrangeModel**(*init=None*, *wrange_selector=None*, *wrange=None*, *order=None*, *extract_orders=None*, *wunits=None*, ***kwargs*)
A model for a reference file of type "wavelengthrange". The model is used by MIRI, NIRSPEC, NIRCAM, and NIRISS

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

class jwst.datamodels.**WfssBkgModel**(*init=None*, ***kwargs*)
A data model for 2D WFSS master background reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D.
- **dq** (*numpy array*) – The data quality array. 2-D.
- **err** (*numpy array*) – The error array. 2-D.
- **dq_def** (*numpy array*) – The data quality definitions table.

Metadata

Metadata information associated with a data model is accessed through its `meta` member. For example, to access the date that an observation was made:

```
print(model.meta.observation.date)
```

Metadata values are automatically type-checked against the schema when they are set. Therefore, setting a keyword which expects a number to a string will raise an exception:

```
>>> from jwst.datamodels import ImageModel
>>> model = ImageModel()
>>> model.meta.target.ra = "foo"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "site-packages/jwst.datamodels/schema.py", line 672, in __setattr__
    object.__setattr__(self, attr, val)
  File "site-packages/jwst.datamodels/schema.py", line 490, in __set__
    val = self.to_basic_type(val)
  File "site-packages/jwst.datamodels/schema.py", line 422, in to_basic_type
    raise ValueError(e.message)
ValueError: 'foo' is not of type u'number'
```

The set of available metadata elements is defined in a YAML Schema that ships with *jwst.datamodels*.

There is also a utility method for finding elements in the metadata schema. `search_schema` will search the schema for the given substring in metadata names as well as their documentation. The search is case-insensitive:

```
>>> from jwst.datamodels import ImageModel
# Create a model of the desired type
>>> model = ImageModel()
# Call `search_schema` on it to find possibly related elements.
>>> model.search_schema('target')
target: Information about the target
target.dec: DEC of the target
target.name: Standard astronomical catalog name for the target
target.proposer: Proposer's name for the target
target.ra: RA of the target
target.type: Fixed target, moving target, or generic target
```

An alternative method to get and set metadata values is to use a dot-separated name as a dictionary lookup. This is useful for databases, such as CRDS, where the path to the metadata element is most conveniently stored as a string. The following two lines are equivalent:

```
print(model['meta.observation.date'])
print(model.meta.observation.date)
```

Working with lists

Unlike ordinary Python lists, lists in the schema may be restricted to only accept a certain set of values. Items may be added to lists in two ways: by passing a dictionary containing the desired key/value pairs for the object, or using the lists special method `item` to create a metadata object and then assigning that to the list.

For example, suppose the metadata element `meta.transformations` is a list of transformation objects, each of which has a `type` (https://docs.python.org/3/library/functions.html#type) (string) and a `coeff` (number) member. We can assign elements to the list in the following equivalent ways:

```
>>> trans = model.meta.transformations.item()
>>> trans.type = 'SIN'
>>> trans.coeff = 42.0
>>> model.meta.transformations.append(trans)
```

(continues on next page)

(continued from previous page)

```
>>> model.meta.transformations.append({'type': 'SIN', 'coeff': 42.0})
```

When accessing the items of the list, the result is a normal metadata object where the attributes are type-checked:

```
>>> trans = model.meta.transformations[0]
>>> print(trans)
<jwst.datamodels.schema.Transformations object at 0x123a810>
>>> print(trans.type)
SIN
>>> trans.type = 42.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "site-packages/jwst.datamodels/schema.py", line 672, in __setattr__
    object.__setattr__(self, attr, val)
  File "site-packages/jwst.datamodels/schema.py", line 490, in __set__
    val = self.to_basic_type(val)
  File "site-packages/jwst.datamodels/schema.py", line 422, in to_basic_type
    raise ValueError(e.message)
ValueError: 42.0 is not of type u'string'
```

JSON Schema

The *jwst.datamodels* library defines its metadata using Draft 4 of the JSON Schema specification (http://tools.ietf.org/html/draft-zyp-json-schema-04), but jwst.datamodels uses YAML for the syntax. A good resource for learning about JSON schema is the book Understanding JSON Schema (http://spacetelescope.github.com/understanding-json-schema). The mapping from Javascript to Python concepts (such as Javascript "array" == Python "list") is added where applicable.

In addition to the standard JSON Schema keywords, `jwst.datamodels` also supports the following additional keywords.

Arrays

The following keywords have to do with validating n-dimensional arrays:

- `ndim`: The number of dimensions of the array.
- `max_ndim`: The maximum number of dimensions of the array.
- `datatype`: For defining an array, `datatype` should be a string. For defining a table, it should be a list.
- **array**: `datatype` should be one of the following strings, representing fixed-length datatypes:

  bool8, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64, float128, complex64, complex128, complex256

Or, for fixed-length strings, an array `[ascii, XX]` where `XX` is the maximum length of the string.

(Datatypes whose size depend on the platform are not supported since this would make files less portable).

- **table**: `datatype` should be a list of dictionaries. Each element in the list defines a column and has the following keys:
  - `datatype`: A string to select the type of the column. This is the same as the `datatype` for an array (as described above).
  - `name` (optional): An optional name for the column.

  - `shape` (optional): The shape of the data in the column. May be either an integer (for a single-dimensional shape), or a list of integers.

FITS-specific Schema Attributes

jwst.datamodels also adds some new keys to the schema language in order to handle reading and writing FITS files. These attributes all have the prefix `fits_`.

- `fits_keyword`: Specifies the FITS keyword to store the value in. Must be a string with a maximum length of 8 characters.
- `fits_hdu`: Specifies the FITS HDU to store the value in. May be a number (to specify the nth HDU) or a name (to specify the extension with the given `EXTNAME`). By default this is set to 0, and therefore refers to the primary HDU.

Creating a new model

This tutorial describes the steps necessary to define a new model type using *jwst.datamodels*.

For further reading and details, see the reference materials in *Metadata*.

In this tutorial, we'll go through the process of creating a new type of model for a file format used for storing the bad pixel mask for JWST's MIRI instrument. This file format has a 2D array containing a bit field for each of the pixels, and a table describing what each of the bits in the array means.

Note: While an attempt is made to present a real-world example here, it may not reflect the actual final format of this file type, which is still subject to change at the time of this writing.

This example will be built as a third-party Python package, i.e. not part of *jwst.datamodels* itself. Doing so adds a few extra wrinkles to the process, and it's most helpful to show what those wrinkles are. To skip ahead and just see the example in its entirety, see the `examples/custom_model` directory within the *jwst.datamodels* source tree.

Directory layout

The bare minimum directory layout for a Python package that creates a custom model is as below:

```
.
|-- lib
|   |--- __init__.py
|   |--- bad_pixel_mask.py
|   |--- schemas
|   |--- bad_pixel_mask.schema.yaml
|   |--- tests
|       |--- __init__.py
|       |--- test_bad_pixel_mask.py
|       |--- data
|       |--- bad_pixel_mask.fits
|--- setup.py
```

The main pieces are the new schema in `bad_pixel_mask.schema.yaml`, the custom model class in `bad_pixel_mask.py`, a distutils-based `setup.py` file to install the package, and some unit tests and associated data. Normally, you would also have some code that *uses* the custom model included in the package, but that isn't included in this minimal example.

The schema file

Let's start with the schema file, `bad_pixel_mask.schema.yaml`. There are a few things it needs to do:

1) It should contain all of the core metadata from the core schema that ships with *`jwst.datamodels`*. In JSON Schema parlance, this schema "extends" the core schema. In object-oriented programming terminology, this could be said that our schema "inherits from" the core schema. It's all the same thing.
2) Define the pixel array containing the information about each of the bad pixels. This will be an integer for each pixel where each bit is ascribed a particular meaning.
3) Define a table describing what each of the bit fields in the pixel array means. This will have three columns: one for the bit field's number (a power of 2), one for a name token to identify it, and one with a human-readable description.

At the top level, every JSON schema must be a mapping (dictionary) of type "object", and should include the core schema:

```
allOf:
   - $ref: "http://jwst.stsci.edu/schemas/core.schema.yaml"
   - type: object
     properties:
        ...
```

There's a lot going on in this one item. `$ref` declares the schema fragment that we want to include (the "base class" schema). Here, the `$ref` mapping causes the system to go out and fetch the content at the given URL, and then replace the mapping with that content.

The `$ref` URL can be a relative URL, in which case it is relative to the schema file where `$ref` is used. In our case, however, it's an absolute URL. Before you visit that URL to see what's there, I'll save you the trouble: there is nothing at that HTTP address. The host `jwst.stsci.edu` is recognized as a "special" address by the system that causes the schema to be looked up alongside installed Python code. For example, to refer to a (hypothetical) `my_instrument` schema that ships with a Python package called `astroboy`, use the following URL:

```
http://jwst.stsci.edu/schemas/astroboy/my_instrument.schema.yaml
```

The "package" portion may be omitted to refer to schemas in the *`jwst.datamodels`* core, which is how we arrive at the URL we're using here:

```
http://jwst.stsci.edu/schemas/core.schema.yaml
```

Note: At some time in the future, we will actually be hosting schemas at a URL similar to the one above. This will allow schemas to be shared with tools built in languages other than Python. Until we have that hosting established, this works quite well and does not require any coordination among Python packages that define new models. Keep an eye out if you use this feature, though – the precise URL used may change.

The next part of the file describes the array data, that is, things that are Numpy arrays on the Python side and images or tables on the FITS side.

First, we describe the main `"dq"` array. It's declared to be 2-dimensional, and each element is an unsigned 32-bit integer:

```
properties:
  dq:
    title: Bad pixel mask
    fits_hdu: DQ
```

(continues on next page)

(continued from previous page)

```
    default: 0
    ndim: 2
    datatype: uint16
```

The next entry describes a table that will store the mapping between bit fields and their meanings. This table has four columns:

- `BIT`: The value of the bit field (a power of 2)
- `VALUE`: The value resulting when raising 2 to the BIT power
- `NAME`: The name used to refer to the bit field
- `DESCRIPTION`: A longer, human-readable description of the bit field

```
dq_def:
  title: DQ flag definitions
  fits_hdu: DQ_DEF
  dtype:
    - name: BIT
      datatype: uint32
    - name: VALUE
      datatype: uint32
    - name: NAME
      datatype: [ascii, 40]
    - name: DESCRIPTION
      datatype: [ascii, 80]
```

And finally, we add a metadata element that is specific to this format. To avoid recomputing it repeatedly, we'd like to store a sum of all of the "bad" (i.e. non-zero) pixels stored in the bad pixel mask array. In the model, we want to refer to this value as `model.meta.bad_pixel_count`. In the FITS file, lets store this in the primary header in a keyword named `BPCOUNT`:

```
meta:
  properties:
    bad_pixel_count:
      type: integer
      title: Total count of all bad pixels
      fits_keyword: BPCOUNT
```

That's all there is to the schema file, and that's the hardest part.

The model class

Now, let's see how this schema is tied in with a new Python class for the model.

First, we need to import the `DataModel` class, which is the base class for all models:

```
from jwst.datamodels import DataModel
```

Then we create a new Python class that inherits from `DataModel`, and set its `schema_url` class member to point to the schema that we just defined above:

```
class MiriBadPixelMaskModel(DataModel):
    schema_url = "bad_pixel_mask.schema.yaml"
```

Here, the `schema_url` has all of the "magical" URL abilities described above when we used the `$ref` feature. However, here we are using a relative URL. In this case, it is relative to the file in which this class is defined, with a small twist to avoid intermingling Python code and schema files: It looks for the given file in a directory called `schemas` inside the directory containing the Python module in which the class is defined.

As an alternative, we could just as easily have said that we want to use the `image` schema from the core without defining any extra elements, by setting `schema_url` to:

```
schema_url = "http://jwst.stsci.edu/schemas/image.schema.yaml"
```

Note: At this point you may be wondering why both the schema and the class have to inherit from base classes. Certainly, it would have been more convenient to have the inheritance on the Python side automatically create the inheritance on the schema side (or vice versa). The reason we can't is that the schema files are designed to be language-agnostic: it is possible to use them from an entirely different implementation of the *jwst.datamodels* framework possibly even written in a language other than Python. So the schemas need to "stand alone" from the Python classes. It's certainly possible to have the schema inherit from one thing and the Python class inherit from another, and the *jwst.datamodels* framework won't and can't really complain, but doing that is only going to lead to confusion, so just don't do it.

Within this class, we'll define a constructor. All model constructors must take the highly polymorphic `init` value as the first argument. This can be a file, another model, or all kinds of other things. See the docstring of `jwst.datamodels.DataModel.__init__` for more information. But we're going to let the base class handle that anyway.

The rest of the arguments are up to you, but generally it's handy to add a couple of keyword arguments so the user can data arrays when creating a model from scratch. If you don't need to do that, then technically writing a new constructor for the model is optional:

```
def __init__(self, init=None, dq=None, dq_def=None, **kwargs):
    """
    A data model to represent MIRI bad pixel masks.

    Parameters
    ----------
    init : any
        Any of the initializers supported by `~jwst.datamodels.DataModel`.

    dq : numpy array
        The data quality array.

    dq_def : numpy array
        The data quality definitions table.
    """
    super(MiriBadPixelMaskModel, self).__init__(init=init, **kwargs)

    if dq is not None:
        self.dq = dq

    if dq_def is not None:
        self.dq_def = dq_def
```

The `super..` line is just the standard Python way of calling the constructor of the base class. The rest of the constructor sets the arrays on the object if any were provided.

The other methods of your class may provide additional conveniences on top of the underlying file format. This is completely optional and if your file format is supported well enough by the underlying schema alone, it may not be

necessary to define any extra methods.

In the case of our example, it would be nice to have a function that, given the name of a bit field, would return a new array that is `True` (https://docs.python.org/3/library/constants.html#True) wherever that bit field is true in the main mask array. Since the order and content of the bit fields are defined in the `dq_def` table, the function should use it in order to do this work:

```
def get_mask_for_field(self, name):
    """
    Returns an array that is `True` everywhere a given bitfield is
    True in the mask.

    Parameters
    ----------
    name : str
        The name of the bit field to retrieve

    Returns
    -------
    array : boolean numpy array
        `True` everywhere the requested bitfield is `True`.  This
        is the same shape as the mask array.  This array is a copy
        and changes to it will not affect the underlying model.
    """
    # Find the field value that corresponds to the given name
    field_value = None
    for value, field_name, title in self.dq_def:
        if field_name == name:
            field_value = value
            break
    if field_value is None:
        raise ValueError("Field name {0} not found".format(name))

    # Create an array that is `True` only for the requested
    # bit field
    return self.dq & field_value
```

One thing to note here: this array is semantically a "copy" of the underlying data. Most Numpy arrays in the model framework are mutable, and we expect that changing their values will update the model itself, and be saved out by subsequent saves to disk. Since the array we are returning here has no connection back to the model's main data array (`mask`), it's helpful to remind the user of that in the docstring, and not present it as a member or property, but as a getter function.

Note: Since handling bit fields like this is such a commonly useful thing, it's possible that this functionality will become a part of *`jwst.datamodels`* itself in the future. However, this still stands as a good example of something someone may want to do in a custom model class.

Lastly, remember the `meta.bad_pixel_count` element we defined above? We need some way to make sure that whenever the file is written out that it has the correct value. The model may have been loaded and modified. For this, `DataModel` has the `on_save` method hook, which may be overridden by the subclass to add anything that should happen just before saving:

```
def on_save(self, path):
    super(MiriBadPixelMaskModel, self).on_save(path)

    self.meta.bad_pixel_count = np.sum(self.mask != 0)
```

Note that here, like in the constructor, it is important to "chain up" to the base class so that any things that the base class wants to do right before saving also happen.

The `setup.py` script

Writing a distutils `setup.py` script is beyond the scope of this tutorial but it's worth noting one thing. Since the schema files are not Python files, they are not automatically picked up by distutils, and must be included in the `package_data` option. A complete, yet minimal, `setup.py` is presented below:

```
#!/usr/bin/env python

from distutils.core import setup

setup(
    name='custom_model',
    description='Custom model example for jwst.datamodels',
    packages=['custom_model', 'custom_model.tests'],
    package_dir={'custom_model': 'lib'},
    package_data={'custom_model': ['schemas/*.schema.yaml'],
                  'custom_model.tests' : ['data/*.fits']}
    )
```

Using the new model

The new model can now be used. For example, to get the locations of all of the "hot" pixels:

```
from custom_model.bad_pixel_mask import MiriBadPixelMaskModel

with MiriBadPixelMaskModel("bad_pixel_mask.fits") as dm:
    hot_pixels = dm.get_mask_for_field('HOT')
```

A table-based model

In addition to n-dimensional data arrays, models can also contain tabular data. For example, the photometric correction reference file used in the JWST calibration pipeline consists of a table with 7 columns. The schema file for this model looks like this:

```
title: Photometric flux conversion data model
allOf:
  - $ref: "core.schema.yaml"
  - type: object
    properties:
      phot_table:
        title: Photometric flux conversion factors table
        fits_hdu: PHOTOM
        datatype:
          - name: filter
            datatype: [ascii, 12]
          - name: photflam
            datatype: float32
          - name: photerr
            datatype: float32
```

(continues on next page)

(continued from previous page)

```
          - name: nelem
            datatype: int16
          - name: wavelength
            datatype: float32
            shape: [50]
          - name: response
            datatype: float32
            shape: [50]
          - name: resperr
            datatype: float32
            shape: [50]
```

In this particular table the first 4 columns contain scalar entries of types string, float, and integer. The entries in the final 3 columns, on the other hand, contain 1-D float arrays (vectors). The "shape" attribute is used to designate the dimensions of the arrays.

The corressponding python module containing the data model class is quite simple:

```
class PhotomModel(model_base.DataModel):
    """
    A data model for photom reference files.
    """
    schema_url = "photom.schema.json"

    def __init__(self, init=None, phot_table=None, **kwargs):
        super(PhotomModel, self).__init__(init=init, **kwargs)

        if phot_table is not None:
            self.phot_table = phot_table
```

FITS file structures and contents

Here we describe the structure and content of the most frequently used forms of FITS files for JWST science data products. Each type of FITS file is the result of serialization of a corresponding data model.

Common Features

All FITS science products have a few common features to their structure and organization:

1. The 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 occurence 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) should be stored in keywords in the headers of those extensions.
2. All data related to the product are contained in one or more FITS Image or Table extensions. The header of each extension may contain keywords that pertain uniquely to that extension.

Level-1 and Level-2 exposure-based products, which contain the data from an individual exposure on an individual detector, use the following file naming scheme:

```
jw{ppppp}{ooo}{vvv}_{gg}{s}{aa}_{eeeee}_{detector}_{suffix}.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
- detector: detector name (e.g. 'nrca1', 'nrcblong', 'mirimage')
- suffix: product type identifier (e.g. 'uncal', 'rate', 'cal')

An example Level-2a product FITS file name is:

```
jw93065002001_02101_00001_nrca1_rate.fits
```

Specific products

This section lists the organization and contents of each type of science product in FITS form.

Raw Level-1b (suffix = `uncal`)

Exposure raw data (level-1b) products are designated with a file name suffix of "uncal." These files usually contain only the raw pixel values from an exposure, with the addition of a table extension that contains some downlinked meta data pertaining to individual groups. Additional extensions can be included for certain instruments and readout types. If the zero-frame was requested to be downlinked, an additional image extension is included that contains those data. MIRI exposures also contain an additional image extension with the values from the reference output. The FITS file structure is as follows.

HDU	Content	EXTNAME	HDU Type	Data Type	Dimensions
0	Primary header	N/A	N/A	N/A	N/A
1	Pixel values	SCI	IMAGE	uint16	ncols x nrows x ngroups x nints
2	Group meta	GROUP	BINTABLE	N/A	variable
3	Zero frame images	ZEROFRAME	IMAGE	uint16	ncols x nrows x nints
4	Reference output	REFOUT	IMAGE	uint16	ncols x 256 x ngroups x nints

The raw pixel values in the SCI extension are stored as a 4-D data array, having dimensions equal to the 2-D size of the detector readout, with the data from the multiple groups (ngroups) within each integration stored along the 3rd axis, and the multiple integrations (nints) stored along the 4th axis.

If zero-frame data are downlinked, there will be one zero-frame image for each integration, stored as a 3-D cube (each cube plane corresponds to an integration).

Level-2 ramp data (suffix = `ramp`)

As soon as raw level-1b products are loaded into the calibration pipeline the contents of the product is modified to include additional data extensions, as well as converting the raw SCI (and ZEROFRAME and REFOUT, if present) array values from integer to floating-point data type. New data arrays that are added include an ERR extension and two types of data quality flag extensions. There is a 2-D `PIXELDQ` extension that will contain flags that pertain to all groups and all integrations, and there is also a 4-D `GROUPDQ` extension for containing flags that pertain to individual groups within individual integrations. The FITS file layout is as follows:

HDU	Content	EXTNAME	HDU Type	Data Type	Dimensions
0	Primary header	N/A	N/A	N/A	N/A
1	Pixel values	SCI	IMAGE	float32	ncols x nrows x ngroups x nints
2	2-D data quality	PIXELDQ	IMAGE	uint32	ncols x nrows
3	4-D data quality	GROUPDQ	IMAGE	uint8	ncols x nrows x ngroups x nints
4	Error values	ERR	IMAGE	float32	ncols x nrows x ngroups x nints

Any additional extensions that were present in the raw level-1b file (e.g. GROUP, ZEROFRAME, REFOUT) will be carried along and will also appear in the level-2 ramp product.

Level-2a countrate products (suffix = `rate` and `rateints`)

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

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

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

HDU	Content	EXTNAME	HDU Type	Data Type	Dimensions
0	Primary header	N/A	N/A	N/A	N/A
1	Pixel values	SCI	IMAGE	float32	ncols x nrows x nints
2	Data quality	DQ	IMAGE	uint32	ncols x nrows x nints
3	Error values	ERR	IMAGE	float32	ncols x nrows x nints

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

HDU	Content	EXTNAME	HDU Type	Data Type	Dimensions
0	Primary header	N/A	N/A	N/A	N/A
1	Pixel values	SCI	IMAGE	float32	ncols x nrows
2	Data quality	DQ	IMAGE	uint32	ncols x nrows
3	Error values	ERR	IMAGE	float32	ncols x nrows

Note that the two separate forms of PIXELDQ and GROUPDQ flags from the previous types of products have been combined into a single DQ extension with the same dimensions as the SCI and ERR components.

Level-2b calibrated products (suffix = `cal` and `calints`)

Single exposure calibrated products duplicate the format and content of level-2a products. As with level-2a, there are two different forms of calibrated products: one containing results for individual integrations (`calints`) and one for exposure-wide results (`cal`).

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

HDU	Content	EXTNAME	HDU Type	Data Type	Dimensions
0	Primary header	N/A	N/A	N/A	N/A
1	Pixel values	SCI	IMAGE	float32	ncols x nrows x nints
2	Data quality	DQ	IMAGE	uint32	ncols x nrows x nints
3	Error values	ERR	IMAGE	float32	ncols x nrows x nints

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

HDU	Content	EXTNAME	HDU Type	Data Type	Dimensions
0	Primary header	N/A	N/A	N/A	N/A
1	Pixel values	SCI	IMAGE	float32	ncols x nrows
2	Data quality	DQ	IMAGE	uint32	ncols x nrows
3	Error values	ERR	IMAGE	float32	ncols x nrows

jwst.datamodels Package

Functions

open([init, extensions])	Creates a DataModel from a number of different types

open

jwst.datamodels.**open**(*init=None*, *extensions=None*, ***kwargs*)

Creates a DataModel from a number of different types

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList,*) – numpy array, dict, None
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS , JSON or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given `HDUList`
  - A numpy array: A new model with the data array initialized to what was passed in.
  - dict: The object model tree for the data model
- **extensions** (*list of AsdfExtension*) – A list of extensions to the ASDF to support when reading and writing ASDF files.

Returns **model**

Return type DataModel instance

Classes

DataModel([init, schema, extensions, ...])	Base class of all of the data models.
AmiLgModel([init, schema, extensions, ...])	A data model for AMI LG analysis results.
AsnModel([init])	A data model for association tables.
BarshadowModel([init])	A data model for Bar Shadow correction information.
CameraModel([init, model, input_units, ...])	A model for a reference file of type "camera".
CollimatorModel([init, model, input_units, ...])	A model for a reference file of type "collimator".
CombinedSpecModel([init, schema, ...])	A data model for combined 1D spectra.
ContrastModel([init, schema, extensions, ...])	A data model for coronagraphic contrast curve files.
CubeModel([init])	A data model for 3D image cubes.
DarkModel([init])	A data model for dark reference files.
DarkMIRIModel([init])	A data model for dark MIRI reference files.
DisperserModel([init, angle, gwa_tiltx, ...])	A model for a NIRSPEC reference file of type "disperser".
DistortionModel([init, model, input_units, ...])	A model for a reference file of type "distortion".
DistortionMRSModel([init, x_model, y_model, ...])	A model for a reference file of type "distortion" for the MIRI MRS.
DrizProductModel([init, schema, extensions, ...])	A data model for drizzle-generated products.
DrizParsModel([init])	A data model for drizzle parameters reference tables.
Extract1dImageModel([init, schema, ...])	A data model for the extract_1d reference image array.
FilteroffsetModel([init, filters])	A model for a NIRSPEC reference file of type "disperser".
FlatModel([init])	A data model for 2D flat-field images.
NRSFlatModel([init])	A base class for NIRSpec flat-field reference file models.
NirspecFlatModel([init])	A data model for NIRSpec flat-field reference files.
NirspecQuadFlatModel([init])	A data model for NIRSpec flat-field files that differ by quadrant.
FOREModel([init, model, input_units, ...])	A model for a reference file of type "fore".
FPAModel([init, nrs1_model, nrs2_model])	A model for a NIRSPEC reference file of type "fpa".
FringeModel([init])	A data model for 2D fringe correction images.
GainModel([init])	A data model for 2D gain.
GLS_RampFitModel([init, schema, extensions, ...])	A data model for the optional output of the ramp fitting step for the GLS algorithm.
GuiderRawModel([init])	A data model for FGS pipeline input files
GuiderCalModel([init])	A data model for FGS pipeline output files
IFUCubeModel([init])	A data model for 3D IFU cubes.
IFUCubeParsModel([init])	A data model for IFU Cube parameters reference tables.
NirspecIFUCubeParsModel([init])	A data model for Nirspec ifucubepars reference files.
MiriIFUCubeParsModel([init])	A data model for MIRI mrs ifucubepars reference files.
IFUFOREModel([init, model, input_units, ...])	A model for a NIRSPEC reference file of type "ifufore".
IFUImageModel([init])	A data model for 2D IFU images.
IFUPostModel([init, slice_models])	A model for a NIRSPEC reference file of type "ifupost".
IFUSlicerModel([init, model, data])	A model for a NIRSPEC reference file of type "ifuslicer".
ImageModel([init])	A data model for 2D images.
IPCModel([init])	A data model for IPC kernel checking information.
IRS2Model([init, schema, extensions, ...])	A data model for the IRS2 refpix reference file.
LastFrameModel([init])	A data model for Last frame correction reference files.
Level1bModel([init])	A data model for raw 4D ramps level-1b products.
LinearityModel([init])	A data model for linearity correction information.
MaskModel([init])	A data model for 2D masks.

Continued on next page

Table 62 – continued from previous page

ModelContainer([init, persist])	A container for holding DataModels.
MSAModel([init, models, data])	A model for a NIRSPEC reference file of type "msa".
MultiExposureModel([init])	A data model for multi-slit images derived from numerous exposures.
MultiExtract1dImageModel([init])	A data model for extract_1d reference images.
MultiProductModel([init])	A data model for multi-DrizProduct images.
MultiSlitModel([init])	A data model for multi-slit images.
MultiSpecModel([init])	A data model for multi-spec images.
OTEModel([init, model, input_units, ...])	A model for a reference file of type "ote".
NIRCAMGrismModel([init, displ, dispx, ...])	A model for a reference file of type "specwcs" for NIRCAM grisms.
NIRISSGrismModel([init, displ, dispx, ...])	A model for a reference file of type "specwcs" for NIRISS grisms.
OutlierParsModel([init])	A data model for outlier detection parameters reference tables.
PathlossModel([init])	A data model for pathloss correction information.
PersistenceSatModel([init])	A data model for the persistence saturation value (full well).
PixelAreaModel([init])	A data model for the pixel area map
NirspecSlitAreaModel([init])	A data model for the NIRSpec fixed-slit pixel area reference file
NirspecMosAreaModel([init])	A data model for the NIRSpec MOS pixel area reference file
NirspecIfuAreaModel([init])	A data model for the NIRSpec IFU pixel area reference file
PhotomModel([init])	A base class for photometric reference file models.
FgsPhotomModel([init])	A data model for FGS photom reference files.
MiriImgPhotomModel([init])	A data model for MIRI imaging photom reference files.
MiriMrsPhotomModel([init])	A data model for MIRI MRS photom reference files.
NircamPhotomModel([init])	A data model for NIRCam photom reference files.
NirissPhotomModel([init])	A data model for NIRISS photom reference files.
NirspecPhotomModel([init])	A data model for NIRSpec imaging, IFU, and MOS photom reference files.
NirspecFSPhotomModel([init])	A data model for NIRSpec Fixed-Slit (FS) photom reference files.
PsfMaskModel([init])	A data model for coronagraphic 2D PSF mask reference files
QuadModel([init])	A data model for 4D image arrays.
RampModel([init])	A data model for 4D ramps.
MIRIRampModel([init])	A data model for MIRI ramps.
RampFitOutputModel([init, schema, ...])	A data model for the optional output of the ramp fitting step.
ReadnoiseModel([init])	A data model for 2D readnoise.
ReferenceFileModel([init])	A data model for reference tables
ReferenceCubeModel([init])	A data model for 3D reference images
ReferenceImageModel([init])	A data model for 2D reference images
ReferenceQuadModel([init])	A data model for 4D reference images
RegionsModel([init, regions])	A model for a reference file of type "regions".
ResetModel([init])	A data model for reset correction reference files.
ResolutionModel([init])	A data model for Spectral Resolution parameters reference tables.

Continued on next page

Table 62 – continued from previous page

MiriResolutionModel([init])	A data model for MIRI Resolution reference files.
RSCDModel([init])	A data model for the RSCD reference file.
SaturationModel([init])	A data model for saturation checking information.
SlitDataModel([init])	A data model for 2D images.
SlitModel([init])	A data model for 2D images.
SpecModel([init, schema, extensions, ...])	A data model for 1D spectra.
SourceModelContainer([init])	A container to make MultiExposureModel look like ModelContainer
StrayLightModel([init])	A data model for 2D straylight mask.
SuperBiasModel([init])	A data model for 2D super-bias images.
SpecwcsModel([init, model, input_units, ...])	A model for a reference file of type "specwcs".
ThroughputModel([init])	A data model for filter throughput.
TrapDensityModel([init])	A data model for the trap density of a detector, for persistence.
TrapParsModel([init])	A data model for trap capture and decay parameters.
TrapsFilledModel([init, schema, extensions, ...])	A data model for the number of traps filled for a detector, for persistence.
TsoPhotModel([init, radii])	A model for a reference file of type "tsophot".
WavelengthrangeModel([init, ...])	A model for a reference file of type "wavelengthrange".
WaveCorrModel([init, apertures])	
WfssBkgModel([init])	A data model for 2D WFSS master background reference files.

DataModel

class jwst.datamodels.**DataModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: jwst.datamodels.properties.ObjectNode, jwst.datamodels.ndmodel.NDModel

Base class of all of the data models.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.

- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

history	Get the history as a list of entries
schema	
schema_url	
shape	

Methods Summary

add_schema_entry(position, new_schema)	Extend the model's schema by placing the given new_schema at the given dot-separated position in the tree.
clone(target, source[, deepcopy, memo])	
close()	
copy([memo])	Returns a deep copy of this model.
extend_schema(new_schema)	Extend the model's schema using the given schema, by combining it in an "allOf" array.
find_fits_keyword(keyword[, return_result])	Utility function to find a reference to a FITS keyword in this model's schema.
from_asdf(init[, schema])	Load a data model from a ASDF file.
from_fits(init[, schema])	Load a model from a FITS file.
get_envar(name, value)	
get_fileext()	
get_fits_wcs([hdu_name, hdu_ver, key])	Get a `astropy.wcs.WCS` object created from the FITS WCS information in the model.
get_item_as_json_value(key)	Equivalent to __getitem__, except returns the value as a JSON basic type, rather than an arbitrary Python type.
get_primary_array_name()	Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created.
get_resolver(asdf_file)	
get_section(name)	
info()	Return datatype and dimension for each array or table
items()	Iterates over all of the schema items in a flat way.
iteritems()	Iterates over all of the schema items in a flat way.

Continued on next page

Table 64 – continued from previous page

iterkeys()	Iterates over all of the schema keys in a flat way.
itervalues()	Iterates over all of the schema values in a flat way.
keys()	Iterates over all of the schema keys in a flat way.
my_attribute(attr)	Test if attribute is part of the NDData interface
on_save([path])	This is a hook that is called just before saving the file.
read([init, schema, extensions, ...])	**param init**
save(path[, dir_path])	Save to either a FITS or ASDF file, depending on the path.
search_schema(substring)	Utility function to search the metadata schema for a particular phrase.
set_fits_wcs(wcs[, hdu_name])	Sets the FITS WCS information on the model using the given `astropy.wcs.WCS` object.
to_asdf(init, *args, **kwargs)	Write a DataModel to an ASDF file.
to_fits(init, *args, **kwargs)	Write a DataModel to a FITS file.
to_flat_dict([include_arrays])	Returns a dictionary of all of the schema items as a flat dictionary.
update(d[, only])	Updates this model with the metadata elements from another model.
validate()	Re-validate the model instance againsst its schema
validate_required_fields()	Walk the schema and make sure all required fields are in the model
values()	Iterates over all of the schema values in a flat way.
write(path, *args, **kwargs)	

Attributes Documentation

history
Get the history as a list of entries

schema

schema_url = 'core.schema.yaml'

shape

Methods Documentation

add_schema_entry(*position*, *new_schema*)
Extend the model's schema by placing the given new_schema at the given dot-separated position in the tree.

Parameters

- **position** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) –
- **new_schema** (*schema tree*) –

static clone(*target*, *source*, *deepcopy=False*, *memo=None*)

close()

copy(*memo=None*)
Returns a deep copy of this model.

extend_schema(*new_schema*)
Extend the model's schema using the given schema, by combining it in an "allOf" array.

Parameters **new_schema** (*schema tree*) –

find_fits_keyword(*keyword*, *return_result=True*)
Utility function to find a reference to a FITS keyword in this model's schema. This is intended for interactive use, and not for use within library code.

Parameters **keyword** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A FITS keyword name

Returns **locations** – If `return_result` is `True` (https://docs.python.org/3/library/constants.html#True), a list of the locations in the schema where this FITS keyword is used. Each element is a dot-separated path.

Return type list of str

Example

```
>>> model.find_fits_keyword('DATE-OBS')
['observation.date']
```

classmethod from_asdf(*init*, *schema=None*)
Load a data model from a ASDF file.

Parameters

- **init** (*file path, file object, asdf.AsdfFile object*) –
  - file path: Initialize from the given file
  - readable file object: Initialize from the given file object
  - asdf.AsdfFile: Initialize from the given `AsdfFile`.
- **schema** – Same as for `__init__`

Returns **model**

Return type DataModel instance

classmethod from_fits(*init*, *schema=None*)
Load a model from a FITS file.

Parameters

- **init** (*file path, file object, astropy.io.fits.HDUList*) –
  - file path: Initialize from the given file
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given `HDUList`.
- **schema** – Same as for `__init__`

Returns **model**

Return type DataModel instance

get_envar(*name*, *value*)

get_fileext()

get_fits_wcs(*hdu_name='SCI'*, *hdu_ver=1*, *key=' '*)
Get a `astropy.wcs.WCS` object created from the FITS WCS information in the model.

Note that modifying the returned WCS object will not modify the data in this model. To update the model, use *set_fits_wcs*.

Parameters

- **hdu_name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the HDU to get the WCS from. This must use named HDU's, not numerical order HDUs. To get the primary HDU, pass `'PRIMARY'`.
- **key** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of a particular WCS transform to use. This may be either `' '` or `'A'-'Z'` and corresponds to the `"a"` part of the `CTYPEia` cards. *key* may only be provided if *header* is also provided.
- **hdu_ver** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – The extension version. Used when there is more than one extension with the same name. The default value, 1, is the first.

Returns **wcs** – The type will depend on what libraries are installed on this system.

Return type `astropy.wcs.WCS` or `pywcs.WCS` object

get_item_as_json_value(*key*)
Equivalent to __getitem__, except returns the value as a JSON basic type, rather than an arbitrary Python type.

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

get_resolver(*asdf_file*)

get_section(*name*)

info()
Return datatype and dimension for each array or table

items()
Iterates over all of the schema items in a flat way.

Each element is a pair (`key`, `value`). Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as:

```
("meta.observation.date": "2012-04-22T03:22:05.432")
```

iteritems()
Iterates over all of the schema items in a flat way.

Each element is a pair (`key`, `value`). Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as:

```
("meta.observation.date": "2012-04-22T03:22:05.432")
```

iterkeys()
Iterates over all of the schema keys in a flat way.

Each result of the iterator is a `key`. Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as the string `"meta.observation.date"`.

itervalues()
Iterates over all of the schema values in a flat way.

keys()
Iterates over all of the schema keys in a flat way.

Each result of the iterator is a `key`. Each `key` is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the result as the string `"meta.observation.date"`.

my_attribute(*attr*)
Test if attribute is part of the NDData interface

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

read(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –

- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

save (*path*, *dir_path=None*, **args*, ***kwargs*)
Save to either a FITS or ASDF file, depending on the path.

Parameters

- **path** (*string or func*) – File path to save to. If function, it takes one argument with is model.meta.filename and returns the full path string.
- **dir_path** (*string*) – Directory to save to. If not None, this will override any directory information in the `path`

Returns **output_path** – The file path the model was saved in.

Return type str (https://docs.python.org/3/library/stdtypes.html#str)

search_schema (*substring*)
Utility function to search the metadata schema for a particular phrase.

This is intended for interactive use, and not for use within library code.

The searching is case insensitive.

Parameters **substring** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The substring to search for.

Returns **locations**

Return type list of tuples

set_fits_wcs (*wcs*, *hdu_name='SCI'*)
Sets the FITS WCS information on the model using the given `astropy.wcs.WCS` object.

Note that the "key" of the WCS is stored in the WCS object itself, so it can not be set as a parameter to this method.

Parameters

- **wcs** (`astropy.wcs.WCS` or `pywcs.WCS` object) – The object containing FITS WCS information
- **hdu_name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The name of the HDU to set the WCS from. This must use named HDU's, not numerical order HDUs. To set the primary HDU, pass `'PRIMARY'`.

to_asdf (*init*, **args*, ***kwargs*)
Write a DataModel to an ASDF file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `asdf.AsdfFile.write_to`.

to_fits (*init*, **args*, ***kwargs*)
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

to_flat_dict(*include_arrays=True*)
Returns a dictionary of all of the schema items as a flat dictionary.

Each dictionary key is a dot-separated name. For example, the schema element `meta.observation.date` will end up in the dictionary as:

```
{ "meta.observation.date": "2012-04-22T03:22:05.432" }
```

update(*d*, *only=''*)
Updates this model with the metadata elements from another model.

Parameters

- **d** (*model or dictionary-like object*) – The model to copy the metadata elements from. Can also be a dictionary or dictionary of dictionaries or lists.
- **only** (*only update the named hdu from extra_fits, e.g.*) – only='PRIMARY'. Can either be a list of hdu names or a single string. If left blank, update all the hdus.

validate()
Re-validate the model instance againsst its schema

validate_required_fields()
Walk the schema and make sure all required fields are in the model

values()
Iterates over all of the schema values in a flat way.

write(*path*, **args*, ***kwargs*)

AmiLgModel

class jwst.datamodels.**AmiLgModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for AMI LG analysis results.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.

- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

schema_url

Methods Summary

get_primary_array_name()	Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created.

Attributes Documentation

schema_url = 'amilg.schema.yaml'

Methods Documentation

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

AsnModel

class jwst.datamodels.**AsnModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for association tables.

Attributes Summary

schema_url
supported_formats

Methods Summary

parse_table()

Attributes Documentation

schema_url = 'asn.schema.yaml'

supported_formats = ['yaml', 'json', 'fits']

Methods Documentation

parse_table()

BarshadowModel

class jwst.datamodels.**BarshadowModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for Bar Shadow correction information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – Array defining the bar shadow correction as a function of Y and wavelength.
- **variance** (*numpy array*) – Variance array.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'barshadow.schema.yaml'

CameraModel

class jwst.datamodels.**CameraModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
Bases: jwst.datamodels.wcs_ref_models._SimpleModel

A model for a reference file of type “camera”.

Attributes Summary

reftype
schema_url

Methods Summary

populate_meta()	Subclasses can overwrite this to populate specific meta keywords.

Attributes Documentation

reftype = 'camera'

schema_url = 'camera.schema.yaml'

Methods Documentation

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

CollimatorModel

class jwst.datamodels.**CollimatorModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
Bases: jwst.datamodels.wcs_ref_models._SimpleModel

A model for a reference file of type "collimator".

Attributes Summary

reftype	
schema_url	

Methods Summary

populate_meta()	Subclasses can overwrite this to populate specific meta keywords.

Attributes Documentation

reftype = 'collimator'

schema_url = 'collimator.schema.yaml'

Methods Documentation

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

CombinedSpecModel

class jwst.datamodels.**CombinedSpecModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for combined 1D spectra.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'combinedspec.schema.yaml'

ContrastModel

class jwst.datamodels.**ContrastModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for coronagraphic contrast curve files.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'contrast.schema.yaml'

CubeModel

class jwst.datamodels.**CubeModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for 3D image cubes.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 3-D.
- **dq** (*numpy array*) – The data quality array. 3-D.
- **err** (*numpy array*) – The error array. 3-D
- **zeroframe** (*numpy array*) – The zero-frame array. 3-D
- **relsens** (*numpy array*) – The relative sensitivity array.
- **int_times** (*table*) – The int_times table
- **area** (*numpy array*) – The pixel area array. 2-D
- **wavelength** (*numpy array*) – The wavelength array. 2-D
- **var_poisson** (*numpy array*) – The variance due to Poisson noise array. 3-D
- **var_rnoise** (*numpy array*) – The variance due to read noise array. 3-D

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'cube.schema.yaml'

DarkModel

class jwst.datamodels.**DarkModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for dark reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'dark.schema.yaml'

DarkMIRIModel

class jwst.datamodels.**DarkMIRIModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for dark MIRI reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data (integration dependent)
- **dq** (*numpy array*) – The data quality array. (integration dependent)
- **err** (*numpy array (integration dependent)*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'darkMIRI.schema.yaml'

DisperserModel

class jwst.datamodels.**DisperserModel**(*init=None*, *angle=None*, *gwa_tiltx=None*, *gwa_tilty=None*, *kcoef=None*, *lcoef=None*, *tcoef=None*, *pref=None*, *tref=None*, *theta_x=None*, *theta_y=None*, *theta_z=None*, *groovedensity=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A model for a NIRSPEC reference file of type “disperser”.

Attributes Summary

reftype
schema_url

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'disperser'

schema_url = 'disperser.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

DistortionModel

class jwst.datamodels.**DistortionModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
Bases: `jwst.datamodels.wcs_ref_models._SimpleModel`

A model for a reference file of type "distortion".

Attributes Summary

reftype	
schema_url	

Methods Summary

validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'distortion'

schema_url = 'distortion.schema.yaml'

Methods Documentation

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

DistortionMRSModel

class jwst.datamodels.**DistortionMRSModel**(*init=None*, *x_model=None*, *y_model=None*, *alpha_model=None*, *beta_model=None*, *bzero=None*, *bdel=None*, *input_units=None*, *output_units=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a reference file of type "distortion" for the MIRI MRS.

Attributes Summary

reftype	
schema_url	

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'distortion'

schema_url = 'distortion_mrs.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

DrizProductModel

class jwst.datamodels.**DrizProductModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for drizzle-generated products.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.

- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== =============** (*=========*) – Format Read Write Auto-identify
- **==== ===== =============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== =============** –

Attributes Summary

hdrtab
schema_url

Attributes Documentation

hdrtab

schema_url = 'drizproduct.schema.yaml'

DrizParsModel

class jwst.datamodels.**DrizParsModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A data model for drizzle parameters reference tables.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'drizpars.schema.yaml'

Extract1dImageModel

class jwst.datamodels.**Extract1dImageModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for the extract_1d reference image array.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – An array of values that define the extraction regions.
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'extract1dimage.schema.yaml'

FilteroffsetModel

class jwst.datamodels.**FilteroffsetModel**(*init=None*, *filters=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a NIRSPEC reference file of type "disperser".

Attributes Summary

reftype	
schema_url	

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'filteroffset'

schema_url = 'filteroffset.schema.yaml'

Methods Documentation

on_save(*path=None*)

This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

validate()

Convenience function to be run when files are created. Checks that required reference file keywords are set.

FlatModel

class jwst.datamodels.**FlatModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D flat-field images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D.
- **dq** (*numpy array*) – The data quality array. 2-D.
- **err** (*numpy array*) – The error array. 2-D.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'flat.schema.yaml'

NRSFlatModel

class jwst.datamodels.**NRSFlatModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A base class for NIRSpec flat-field reference file models.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec.flat.schema.yaml'

NirspecFlatModel

class jwst.datamodels.**NirspecFlatModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.NRSFlatModel*

A data model for NIRSpec flat-field reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D or 3-D.
- **dq** (*numpy array*) – The data quality array. 2-D or 3-D.
- **err** (*numpy array*) – The error array. 2-D or 3-D.
- **wavelength** (*numpy array*) – The wavelength for each plane of the `data` array. This will only be needed if `data` is 3-D.
- **flat_table** (*numpy array*) – A table of wavelengths and flat-field values, to specify the component of the flat field that can vary over a relatively short distance (can be pixel-to-pixel).

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec_flat.schema.yaml'

NirspecQuadFlatModel

class jwst.datamodels.**NirspecQuadFlatModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.NRSFlatModel*

A data model for NIRSpec flat-field files that differ by quadrant.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D or 3-D.
- **dq** (*numpy array*) – The data quality array. 2-D or 3-D.
- **err** (*numpy array*) – The error array. 2-D or 3-D.
- **wavelength** (*numpy array*) – The wavelength for each plane of the data array. This will only be needed if data is 3-D.
- **flat_table** (*numpy array*) – A table of wavelengths and flat-field values, to specify the component of the flat field that can vary over a relatively short distance (can be pixel-to-pixel).
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec_quad_flat.schema.yaml'

FOREModel

class jwst.datamodels.**FOREModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)
Bases: jwst.datamodels.wcs_ref_models._SimpleModel

A model for a reference file of type “fore”.

Attributes Summary

reftype
schema_url

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	Subclasses can overwrite this to populate specific meta keywords.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'fore'

schema_url = 'fore.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()
Subclasses can overwrite this to populate specific meta keywords.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

FPAModel

class jwst.datamodels.**FPAModel**(*init=None*, *nrs1_model=None*, *nrs2_model=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A model for a NIRSPEC reference file of type "fpa".

Attributes Summary

reftype	
schema_url	

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.

Continued on next page

Table 97 – continued from previous page

validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'fpa'

schema_url = 'fpa.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

FringeModel

class jwst.datamodels.**FringeModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D fringe correction images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'fringe.schema.yaml'

GainModel

class jwst.datamodels.**GainModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D gain.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The 2-D gain array

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'gain.schema.yaml'

GLS_RampFitModel

class jwst.datamodels.**GLS_RampFitModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for the optional output of the ramp fitting step for the GLS algorithm.

Parameters

- **init** (*shape tuple*, *file path*, *file object*, *astropy.io.fits.HDUList*, *numpy array*, *None* (https://docs.python.org/3/library/constants.html#None)) –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model

- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'gls_rampfit.schema.yaml'

GuiderRawModel

class jwst.datamodels.**GuiderRawModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for FGS pipeline input files

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 4-D
- **dq** (*numpy array*) – The data quality array. 2-D.
- **err** (*numpy array*) – The error array. 4-D.
- **plan_star_table** (*table*) – The planned reference star table
- **flight_star_table** (*table*) – The flight reference star table
- **pointing_table** (*table*) – The pointing table
- **centroid_table** (*table*) – The centroid packet table
- **track_sub_table** (*table*) – The track subarray table

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'guider_raw.schema.yaml'

GuiderCalModel

class jwst.datamodels.**GuiderCalModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for FGS pipeline output files

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 3-D
- **dq** (*numpy array*) – The data quality array. 2-D
- **err** (*numpy array*) – The error array. 3-D
- **plan_star_table** (*table*) – The planned reference star table
- **flight_star_table** (*table*) – The flight reference star table
- **pointing_table** (*table*) – The pointing table
- **centroid_table** (*table*) – The centroid packet table
- **track_sub_table** (*table*) – The track subarray table

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'guider_cal.schema.yaml'

IFUCubeModel

class jwst.datamodels.**IFUCubeModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for 3D IFU cubes.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 3-D.
- **dq** (*numpy array*) – The data quality array. 3-D.

- **err** (*numpy array*) – The error array. 3-D
- **weightmap** (*numpy array*) – The weight map array. 3-D
- **wavetable** (*1-D table*) – Optional table of wavelengths of IFUCube slices

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'ifucube.schema.yaml'

IFUCubeParsModel

class jwst.datamodels.**IFUCubeParsModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for IFU Cube parameters reference tables.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'ifucubepars.schema.yaml'

NirspecIFUCubeParsModel

class jwst.datamodels.**NirspecIFUCubeParsModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for Nirspec ifucubepars reference files.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec_ifucubepars.schema.yaml'

MiriIFUCubeParsModel

class jwst.datamodels.**MiriIFUCubeParsModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for MIRI mrs ifucubepars reference files.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'miri_ifucubepars.schema.yaml'

IFUFOREModel

class jwst.datamodels.**IFUFOREModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)

Bases: jwst.datamodels.wcs_ref_models._SimpleModel

A model for a NIRSPEC reference file of type "ifufore".

Attributes Summary

reftype
schema_url

Methods Summary

populate_meta()	Subclasses can overwrite this to populate specific meta keywords.

Attributes Documentation

reftype = 'ifufore'

schema_url = 'ifufore.schema.yaml'

Methods Documentation

populate_meta()

Subclasses can overwrite this to populate specific meta keywords.

IFUImageModel

class jwst.datamodels.**IFUImageModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for 2D IFU images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.

- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **relsens2d** (*numpy array*) – The relative sensitivity 2D array.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'ifuimage.schema.yaml'

IFUPostModel

class jwst.datamodels.**IFUPostModel**(*init=None*, *slice_models=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a NIRSPEC reference file of type "ifupost".

Parameters

- **init** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A file name.
- **slice_models** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – A dictionary with slice transforms with the following entries: {"slice_N": {'linear': astropy.modeling.Model,

  'xpoly': astropy.modeling.Model, 'xpoly_distortion': astropy.modeling.Model, 'ypoly': astropy.modeling.Model, 'ypoly_distortion': astropy.modeling.Model, }

  }

Attributes Summary

reftype
schema_url

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'ifupost'

schema_url = 'ifupost.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

IFUSlicerModel

class jwst.datamodels.**IFUSlicerModel**(*init=None*, *model=None*, *data=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A model for a NIRSPEC reference file of type "ifuslicer".

Attributes Summary

reftype	
schema_url	

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'ifuslicer'

schema_url = 'ifuslicer.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

ImageModel

class jwst.datamodels.**ImageModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for 2D images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **kwargs** (*numpy array*) – The name and value of any array mentioned in the schema to be initialized through the function call.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'image.schema.yaml'

IPCModel

class jwst.datamodels.**IPCModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A data model for IPC kernel checking information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The deconvolution kernel (a very small image).

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'ipc.schema.yaml'

IRS2Model

class jwst.datamodels.**IRS2Model**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for the IRS2 refpix reference file.

Parameters

- **init** (*shape tuple*, *file path*, *file object*, *astropy.io.fits.HDUList*, *numpy array*, *None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **irs2_table** (*numpy array*) – A table with 8 columns and 2916352 (2048 * 712 * 2) rows. All values are float, but these are interpreted as alternating real and imaginary parts (real, imag, real, imag, ...) of complex values. There are four columns for ALPHA and four for BETA.
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model

- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'irs2.schema.yaml'

LastFrameModel

class jwst.datamodels.**LastFrameModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A data model for Last frame correction reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'lastframe.schema.yaml'

Level1bModel

class jwst.datamodels.**Level1bModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for raw 4D ramps level-1b products.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data
- **zeroframe** (*numpy array*) – The zero-frame data
- **refout** (*numpy array*) – The MIRI reference output data
- **group** (*table*) – The group parameters table
- **int_times** (*table*) – The int_times table

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'level1b.schema.yaml'

LinearityModel

class jwst.datamodels.**LinearityModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for linearity correction information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **coeffs** (*numpy array*) – Coefficients defining the nonlinearity function.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Methods Summary

get_primary_array_name()	Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created.

Attributes Documentation

schema_url = 'linearity.schema.yaml'

Methods Documentation

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

MaskModel

class jwst.datamodels.**MaskModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D masks.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url	

Methods Summary

get_primary_array_name()	Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created.

Attributes Documentation

schema_url = 'mask.schema.yaml'

Methods Documentation

get_primary_array_name()
Returns the name "primary" array for this model, which controls the size of other arrays that are implicitly created. This is intended to be overridden in the subclasses if the primary array's name is not "data".

ModelContainer

class jwst.datamodels.**ModelContainer**(*init=None*, *persist=True*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A container for holding DataModels.

This functions like a list for holding DataModel objects. It can be iterated through like a list, DataModels within the container can be addressed by index, and the datamodels can be grouped into a list of lists for grouped looping, useful for NIRCam where grouping together all detectors of a given exposure is useful for some pipeline steps.

Parameters

- **init** (*file path, list of DataModels, or None* (https://docs.python.org/3/library/constants.html#None)) –
  - file path: initialize from an association table
  - list: a list of DataModels of any type
  - None: initializes an empty *ModelContainer* instance, to which DataModels can be added via the `append()` method.
- **persist** (*boolean. If True, do not close model after opening it*) –

Examples

```
>>> container = datamodels.ModelContainer('example_asn.json')
>>> for dm in container:
...     print(dm.meta.filename)
```

Say the association was a NIRCam dithered dataset. The *models_grouped* attribute is a list of lists, the first index giving the list of exposure groups, with the second giving the individual datamodels representing each detector in the exposure (2 or 8 in the case of NIRCam).

```
>>> total_exposure_time = 0.0
>>> for group in container.models_grouped:
...     total_exposure_time += group[0].meta.exposure.exposure_time
```

```
>>> c = datamodels.ModelContainer()
>>> m = datamodels.open('myfile.fits')
>>> c.append(m)
```

Attributes Summary

group_names	Return list of names for the DataModel groups by exposure.
models_grouped	Returns a list of a list of datamodels grouped by exposure.
schema_url	

Methods Summary

append(model)	
copy([memo])	Returns a deep copy of the models in this model container.
extend(models)	
from_asn(filepath, **kwargs)	Load fits files from a JWST association file.
get_recursively(field)	Returns a list of values of the specified field from meta.
insert(index, model)	
pop([index])	
save([path, dir_path, save_model_func])	Write out models in container to FITS or ASDF.

Attributes Documentation

group_names
Return list of names for the DataModel groups by exposure.

models_grouped
Returns a list of a list of datamodels grouped by exposure.

Data from different detectors of the same exposure will have the same group id, which allows grouping by exposure. The following metadata is used for grouping:

meta.observation.program_number meta.observation.observation_number meta.observation.visit_number meta.observation.visit_group meta.observation.sequence_id meta.observation.activity_id meta.observation.exposure_number

schema_url = 'container.schema.yaml'

Methods Documentation

append(*model*)

copy(*memo=None*)
Returns a deep copy of the models in this model container.

extend(*models*)

from_asn(*filepath*, ***kwargs*)
Load fits files from a JWST association file.

Parameters **filepath** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to an association file.

get_recursively(*field*)
Returns a list of values of the specified field from meta.

insert(*index*, *model*)

pop(*index=-1*)

save(*path=None*, *dir_path=None*, *save_model_func=None*, **args*, ***kwargs*)
Write out models in container to FITS or ASDF.

Parameters

- **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or func or None* (https://docs.python.org/3/library/constants.html#None)) –

- If None, the `meta.filename` is used for each model.
- If a string, the string is used as a root and an index is appended.
- If a function, the function takes the two arguments: the value of model.meta.filename and the `idx` index, returning constructed file name.

- **dir_path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Directory to write out files. Defaults to current working dir. If directory does not exist, it creates it. Filenames are pulled from `meta.filename` of each datamodel in the container.
- **save_model_func** (*func or None* (https://docs.python.org/3/library/constants.html#None)) – Alternate function to save each model instead of the models *save* method. Takes one argument, the model, and keyword argument `idx` for an index.

Returns **output_paths** – List of output file paths of where the models were saved.

Return type [str (https://docs.python.org/3/library/stdtypes.html#str)[, ..]]

MSAModel

class jwst.datamodels.**MSAModel**(*init=None*, *models=None*, *data=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a NIRSPEC reference file of type "msa".

Attributes Summary

reftype	
schema_url	

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'msa'

schema_url = 'msa.schema.yaml'

Methods Documentation

on_save(*path=None*)

This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

MultiExposureModel

class jwst.datamodels.**MultiExposureModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for multi-slit images derived from numerous exposures. The intent is that all slits in this model are of the same source, with each slit representing a separate exposure of that source.

This model has a special member `exposures` that can be used to deal with an entire slit at a time. It behaves like a list:

```
>>> multislit_model.exposures.append(image_model)
>>> multislit_model.exposures[0]
<ImageModel>
```

Also, there is an extra attribute, `meta`. This will contain the meta attribute from the exposure from which each slit has been taken.

See the module `exp_to_source` for the initial creation of these models. This is part of the Level 3 processing of multi-objection observations.

Attributes Summary

core_schema_url
schema_url

Attributes Documentation

core_schema_url = 'core.schema.yaml'

schema_url = 'multiexposure.schema.yaml'

MultiExtract1dImageModel

class jwst.datamodels.**MultiExtract1dImageModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for extract_1d reference images.

This model has a special member `images` that can be used to deal with each image separately. It behaves like a list:

```
>>> multiextr1d_img_model.images.append(ref_image_model)
>>> multiextr1d_img_model.images[0]
<Extract1dImageModelModel>
```

If `init` is a file name or an *Extract1dImageModel* instance, an empty *Extract1dImageModel* will be created and assigned to attribute `images[0]`, and the `data` attribute from the input array or *Extract1dImageModel* will be copied to the first element of `images`.

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'multiextract1d.schema.yaml'

MultiProductModel

class jwst.datamodels.**MultiProductModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for multi-DrizProduct images.

This model has a special member `products` that can be used to deal with each DrizProduct at a time. It behaves like a list:

```
>>> multiprod_model.products.append(image_model)
>>> multislit_model.products[0]
<DrizProductModel>
```

If `init` is a file name or an *DrizProductModel* instance, an empty *DrizProductModel* will be created and assigned to attribute `products[0]`, and the `data`, `wht`, `con`, and `relsens` attributes from the input file or *DrizProductModel* will be copied to the first element of `products`.

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'multiproduct.schema.yaml'

MultiSlitModel

class jwst.datamodels.**MultiSlitModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for multi-slit images.

This model has a special member `slits` that can be used to deal with an entire slit at a time. It behaves like a list:

```
>>> multislit_model.slits.append(image_model)
>>> multislit_model.slits[0]
>>> multislit[0]
<SlitModel>
```

If `init` is a file name or an `ImageModel` or a `SlitModel``instance, an empty ``SlitModel` will be created and assigned to attribute `slits[0]`, and the `data`, `dq`, `err`, `var_rnoise`, `var_poisson``and ``relsens` attributes from the input file or model will be copied to the first element of `slits`.

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'multislit.schema.yaml'

MultiSpecModel

class jwst.datamodels.**MultiSpecModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for multi-spec images.

This model has a special member `spec` that can be used to deal with an entire spectrum at a time. It behaves like a list:

```
>>> multispec_model.spec.append(spec_model)
>>> multispec_model.spec[0]
<SpecModel>
```

If `init` is a *SpecModel* instance, an empty *SpecModel* will be created and assigned to attribute `spec[0]`, and the `spec_table` attribute from the input *SpecModel* instance will be copied to the first element of `spec`. *SpecModel* objects can be appended to the `spec` attribute by using its `append` method.

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

Examples

```
>>> output_model = datamodels.MultiSpecModel()
>>> spec = datamodels.SpecModel()        # for the default data type
>>> for slit in input_model.slits:
>>>     slitname = slit.name
>>>     slitmodel = ExtractModel()
>>>     slitmodel.fromJSONFile(extref, slitname)
>>>     column, wavelength, countrate = slitmodel.extract(slit.data)
>>>     otab = np.array(zip(column, wavelength, countrate),
>>>                     dtype=spec.spec_table.dtype)
>>>     spec = datamodels.SpecModel(spec_table=otab)
>>>     output_model.spec.append(spec)
```

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'multispec.schema.yaml'

OTEModel

class jwst.datamodels.**OTEModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)

Bases: jwst.datamodels.wcs_ref_models._SimpleModel

A model for a reference file of type "ote".

Attributes Summary

reftype
schema_url

Methods Summary

populate_meta()	Subclasses can overwrite this to populate specific meta keywords.

Attributes Documentation

reftype = 'ote'

schema_url = 'ote.schema.yaml'

Methods Documentation

populate_meta()

Subclasses can overwrite this to populate specific meta keywords.

NIRCAMGrismModel

class jwst.datamodels.**NIRCAMGrismModel**(*init=None*, *displ=None*, *dispx=None*, *dispy=None*, *invdispl=None*, *invdispx=None*, *invdispy=None*, *orders=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a reference file of type "specwcs" for NIRCAM grisms.

This reference file contains the models for wave, x, and y polynomial solutions that describe dispersion through the grism

Attributes Summary

reftype	
schema_url	

Methods Summary

populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'specwcs'

schema_url = 'specwcs_nircam_grism.schema.yaml'

Methods Documentation

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

NIRISSGrismModel

class jwst.datamodels.**NIRISSGrismModel**(*init=None*, *displ=None*, *dispx=None*, *dispy=None*, *invdispl=None*, *orders=None*, *fwcpos_ref=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a reference file of type "specwcs" for NIRISS grisms.

Attributes Summary

reftype
schema_url

Methods Summary

populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'specwcs'

schema_url = 'specwcs_niriss_grism.schema.yaml'

Methods Documentation

populate_meta()

to_fits()

Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()

Convenience function to be run when files are created. Checks that required reference file keywords are set.

OutlierParsModel

class jwst.datamodels.**OutlierParsModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A data model for outlier detection parameters reference tables.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'outlierpars.schema.yaml'

PathlossModel

class jwst.datamodels.**PathlossModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for pathloss correction information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **pointsource** (*numpy array*) – Array defining the pathloss parameter for point sources.
- **psvar** (*numpy array*) – Variance array.
- **uniform** (*numpy array*) – Pathloss parameter for uniform illumination

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'pathloss.schema.yaml'

PersistenceSatModel

class jwst.datamodels.**PersistenceSatModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for the persistence saturation value (full well).

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'persat.schema.yaml'

PixelAreaModel

class jwst.datamodels.**PixelAreaModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for the pixel area map

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'pixelarea.schema.yaml'

NirspecSlitAreaModel

class jwst.datamodels.**NirspecSlitAreaModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for the NIRSpec fixed-slit pixel area reference file

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **area_table** (*numpy array*) – A table-like object containing row selection criteria made up of the slit id and the pixel area values associated with the slits.
  - slit_id: str[15]
  - pixarea: float32

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec_area_slit.schema.yaml'

NirspecMosAreaModel

class jwst.datamodels.**NirspecMosAreaModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for the NIRSpec MOS pixel area reference file

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **area_table** (*numpy array*) – A table-like object containing row selection criteria made up of MOS shutter parameters and the pixel area values associated with the shutters.
  - quadrant: int16
  - shutter_x: int16
  - shutter_y: int16
  - pixarea: float32

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec_area_mos.schema.yaml'

NirspecIfuAreaModel

class jwst.datamodels.**NirspecIfuAreaModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for the NIRSpec IFU pixel area reference file

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **area_table** (*numpy array*) – A table-like object containing row selection criteria made up of IFU slice id and the pixel area values associated with the slices.
  - slice_id: int16
  - pixarea: float32

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec_area_ifu.schema.yaml'

PhotomModel

class jwst.datamodels.**PhotomModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A base class for photometric reference file models.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'photom.schema.yaml'

FgsPhotomModel

class jwst.datamodels.**FgsPhotomModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.PhotomModel*

A data model for FGS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[5000]
  - relresponse: float32[5000]

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'fgs_photom.schema.yaml'

MiriImgPhotomModel

class jwst.datamodels.**MiriImgPhotomModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.PhotomModel*

A data model for MIRI imaging photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - subarray: str[15]
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[500]
  - relresponse: float32[500]

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'mirimg_photom.schema.yaml'

MiriMrsPhotomModel

class jwst.datamodels.**MiriMrsPhotomModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.PhotomModel*

A data model for MIRI MRS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – An array-like object containing the pixel-by-pixel conversion values in units of DN / sec / mJy / pixel.
- **err** (*numpy array*) – An array-like object containing the uncertainties in the conversion values, in the same units as the data array.
- **dq** (*numpy array*) – An array-like object containing bit-encoded data quality flags, indicating problem conditions for values in the data array.
- **dq_def** (*numpy array*) – A table-like object containing the data quality definitions table.
- **pixsiz** (*numpy array*) – An array-like object containing pixel-by-pixel size values, in units of square arcseconds (arcsec^2).

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'mirmrs_photom.schema.yaml'

NircamPhotomModel

class jwst.datamodels.**NircamPhotomModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.PhotomModel*

A data model for NIRCam photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - pupil: str[12]
  - order: int16
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[3000]
  - relresponse: float32[3000]

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nircam_photom.schema.yaml'

NirissPhotomModel

class jwst.datamodels.**NirissPhotomModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.PhotomModel*

A data model for NIRISS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - pupil: str[12]
  - order: int16
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[5000]
  - relresponse: float32[5000]

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'niriss_photom.schema.yaml'

NirspecPhotomModel

class jwst.datamodels.**NirspecPhotomModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.PhotomModel*

A data model for NIRSpec imaging, IFU, and MOS photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - grating: str[12]
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[150]
  - relresponse: float32[150]
  - reluncertainty: float32[150]

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspec_photom.schema.yaml'

NirspecFSPhotomModel

class jwst.datamodels.**NirspecFSPhotomModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.PhotomModel*

A data model for NIRSpec Fixed-Slit (FS) photom reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **phot_table** (*numpy array*) – A table-like object containing row selection criteria made up of instrument mode parameters and photometric conversion factors associated with those modes.
  - filter: str[12]
  - grating: str[12]
  - slit: str[12]
  - photmjsr: float32
  - uncertainty: float32
  - nelem: int16
  - wavelength: float32[150]
  - relresponse: float32[150]
  - reluncertainty: float32[150]

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'nirspecfs_photom.schema.yaml'

PsfMaskModel

class jwst.datamodels.**PsfMaskModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for coronagraphic 2D PSF mask reference files

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The 2-D mask array

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'psfmask.schema.yaml'

QuadModel

class jwst.datamodels.**QuadModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for 4D image arrays.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 4-D.
- **dq** (*numpy array*) – The data quality array. 4-D.
- **err** (*numpy array*) – The error array. 4-D

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'quad.schema.yaml'

RampModel

class jwst.datamodels.**RampModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for 4D ramps.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **pixeldq** (*numpy array*) – 2-D data quality array.
- **groupdq** (*numpy array*) – 3-D or 4-D data quality array.
- **err** (*numpy array*) – The error array.

- **group** (*table*) – The group parameters table
- **int_times** (*table*) – The int_times table

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'ramp.schema.yaml'

MIRIRampModel

class jwst.datamodels.**MIRIRampModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.RampModel*

A data model for MIRI ramps. Includes the `refout` array.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **pixeldq** (*numpy array*) – 2-D data quality array.
- **groupdq** (*numpy array*) – 3-D or 4-D data quality array.
- **err** (*numpy array*) – The error array.
- **refout** (*numpy array*) – The array of reference output data.
- **group** (*table*) – The group parameters table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'miri_ramp.schema.yaml'

RampFitOutputModel

class jwst.datamodels.**RampFitOutputModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for the optional output of the ramp fitting step.

In the parameter definitions below, `n_int` is the number of integrations, `max_seg` is the maximum number of segments that were fit, `nreads` is the number of reads in an integration, and `ny` and `nx` are the height and width of the image.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **slope** (*numpy array (n_int, max_seg, ny, nx)*) –
- **sigslope** (*numpy array (n_int, max_seg, ny, nx)*) –
- **var_poisson** (*numpy array (n_int, max_seg, ny, nx)*) –
- **var_rnoise** (*numpy array (n_int, max_seg, ny, nx)*) –
- **yint** (*numpy array (n_int, max_seg, ny, nx)*) –
- **sigyint** (*numpy array (n_int, max_seg, ny, nx)*) –
- **pedestal** (*numpy array (n_int, max_seg, ny, nx)*) –
- **weights** (*numpy array (n_int, max_seg, ny, nx)*) –
- **crmag** (*numpy array (n_int, max_seg, ny, nx)*) –
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'rampfitoutput.schema.yaml'

ReadnoiseModel

class jwst.datamodels.**ReadnoiseModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D readnoise.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – Read noise for all pixels. 2-D.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'readnoise.schema.yaml'

ReferenceFileModel

class jwst.datamodels.**ReferenceFileModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for reference tables

Parameters **init** (*any*) – Any of the initializers supported by *DataModel*.

Attributes Summary

schema_url

Methods Summary

validate()	Convenience function to be run when files are created.

Attributes Documentation

schema_url = 'referencefile.schema.yaml'

Methods Documentation

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

ReferenceCubeModel

class jwst.datamodels.**ReferenceCubeModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 3D reference images

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'referencecube.schema.yaml'

ReferenceImageModel

class jwst.datamodels.**ReferenceImageModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D reference images

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'referenceimage.schema.yaml'

ReferenceQuadModel

class jwst.datamodels.**ReferenceQuadModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 4D reference images

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'referencequad.schema.yaml'

RegionsModel

class jwst.datamodels.**RegionsModel**(*init=None*, *regions=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A model for a reference file of type “regions”.

Attributes Summary

reftype
schema_url

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'regions'

schema_url = 'regions.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

ResetModel

class jwst.datamodels.**ResetModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for reset correction reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'reset.schema.yaml'

ResolutionModel

class jwst.datamodels.**ResolutionModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for Spectral Resolution parameters reference tables.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'resolution.schema.yaml'

MiriResolutionModel

class jwst.datamodels.**MiriResolutionModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ResolutionModel*

A data model for MIRI Resolution reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by '~jwst.datamodels.DataModel'
- **resolving_power_table** (*table*) – A table containing resolving power of the MRS. THe table consist of 11 columns and 12 rows. Each row corresponds to a band. The columns give the name of band, central wavelength, and polynomial coefficeints (a,b,c) needed to obtain the limits and average value of the spectral resolution.
- **psf_fwhm_alpha_table** (*table*) – A table with 5 columns. Column 1 gives the cutoff wavelength where the polynomials describing alpha FWHM change. Columns 2 and 3 give the polynomial cofficients (a,b) describing alpha FWHM for wavelengths shorter than cuttoff. Columns 4 and 5 give the polynomial coefficients (a,b) describing alpha FWHM for wavelengths longer than the cutoff.
- **psf_fwhm_beta_table** (*table*) – A table with 5 columns. Column 1 gives the cutoff wavelength where the polynomials describing alpha FWHM change. Columns 2 and 3 give the polynomial cofficients (a,b) describing beta FWHM for wavelengths shorter than cuttoff. Columns 4 and 5 give the polynomial coefficients (a,b) describing beta FWHM for wavelengths longer than the cutoff.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'miri_resolution.schema.yaml'

RSCDModel

class jwst.datamodels.**RSCDModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for the RSCD reference file.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **rscd_table** (*numpy array*) – A table with seven columns, three string-valued that identify which row to select, and four float columns containing coefficients.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'rscd.schema.yaml'

SaturationModel

class jwst.datamodels.**SaturationModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for saturation checking information.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'saturation.schema.yaml'

SlitDataModel

class jwst.datamodels.**SlitDataModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for 2D images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **relsens** (*numpy array*) – The relative sensitivity table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'slitdata.schema.yaml'

SlitModel

class jwst.datamodels.**SlitModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.DataModel*

A data model for 2D images.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **err** (*numpy array*) – The error array.
- **relsens** (*numpy array*) – The relative sensitivity table.
- **int_times** (*table*) – The int_times table

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'slit.schema.yaml'

SpecModel

class jwst.datamodels.**SpecModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for 1D spectra.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **spec_table** (*numpy array*) – A table with at least four columns: wavelength, flux, an error estimate for the flux, and data quality flags.
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - `astropy.io.fits.HDUList`: Initialize from the given `HDUList`.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'spec.schema.yaml'

SourceModelContainer

class jwst.datamodels.**SourceModelContainer**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ModelContainer*

A container to make MultiExposureModel look like ModelContainer

Methods Summary

save([path, dir_path, save_model_func])	Save out the container as a MultiExposureModel

Methods Documentation

save(*path=None*, *dir_path=None*, *save_model_func=None*, **args*, ***kwargs*)
Save out the container as a MultiExposureModel

StrayLightModel

class jwst.datamodels.**StrayLightModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D straylight mask.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – 2-D straylight mask array.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'straylight.schema.yaml'

SuperBiasModel

class jwst.datamodels.**SuperBiasModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D super-bias images.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'superbias.schema.yaml'

SpecwcsModel

class jwst.datamodels.**SpecwcsModel**(*init=None*, *model=None*, *input_units=None*, *output_units=None*, ***kwargs*)

Bases: jwst.datamodels.wcs_ref_models._SimpleModel

A model for a reference file of type "specwcs".

Attributes Summary

reftype
schema_url

Methods Summary

validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'specwcs'

schema_url = 'specwcs.schema.yaml'

Methods Documentation

validate()

Convenience function to be run when files are created. Checks that required reference file keywords are set.

ThroughputModel

class jwst.datamodels.**ThroughputModel**(*init=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A data model for filter throughput.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'throughput.schema.yaml'

TrapDensityModel

class jwst.datamodels.**TrapDensityModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for the trap density of a detector, for persistence.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data.
- **dq** (*numpy array*) – The data quality array.
- **dq_def** (*numpy array*) – The data quality definitions table.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'trapdensity.schema.yaml'

TrapParsModel

class jwst.datamodels.**TrapParsModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for trap capture and decay parameters.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **trappars_table** (*numpy array*) – A table with three columns for trap-capture parameters and one column for the trap-decay parameter. Each row of the table is for a different trap family.

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'trappars.schema.yaml'

TrapsFilledModel

class jwst.datamodels.**TrapsFilledModel**(*init=None*, *schema=None*, *extensions=None*, *pass_invalid_values=False*, *strict_validation=False*, ***kwargs*)

Bases: *jwst.datamodels.DataModel*

A data model for the number of traps filled for a detector, for persistence.

Parameters

- **init** (*shape tuple, file path, file object, astropy.io.fits.HDUList, numpy array, None* (https://docs.python.org/3/library/constants.html#None)) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The map of the number of traps filled over the detector, with one plane for each "trap family."
- **init** –
  - None: A default data model with no shape
  - shape tuple: Initialize with empty data of the given shape
  - file path: Initialize from the given file (FITS or ASDF)
  - readable file object: Initialize from the given file object
  - astropy.io.fits.HDUList: Initialize from the given HDUList.
  - A numpy array: Used to initialize the data array
  - dict: The object model tree for the data model
- **schema** (*tree of objects representing a JSON schema, or string naming a schema, optional*) – The schema to use to understand the elements on the model. If not provided, the schema associated with this class will be used.
- **extensions** (*classes extending the standard set of extensions, optional.*) – If an extension is defined, the prefix used should be 'url'.
- **pass_invalid_values** (*If true, values that do not validate the schema*) – will be added to the metadata. If false, they will be set to None
- **strict_validation** (*if true, an schema validation errors will generate*) – an excption. If false, they will generate a warning.
- **available built-in formats are** (*The*) –
- **==== ===== ============** (*=========*) – Format Read Write Auto-identify
- **==== ===== ============** –
- **Yes Yes Yes** (*datamodel*) –
- **==== ===== ============** –

Attributes Summary

schema_url

Attributes Documentation

schema_url = 'trapsfilled.schema.yaml'

TsoPhotModel

class jwst.datamodels.**TsoPhotModel**(*init=None*, *radii=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a reference file of type "tsophot".

Attributes Summary

reftype
schema_url

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'tsophot'

schema_url = 'tsophot.schema.yaml'

Methods Documentation

on_save(*path=None*)

This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

to_fits()

Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

WavelengthrangeModel

class jwst.datamodels.**WavelengthrangeModel**(*init=None*, *wrange_selector=None*, *wrange=None*, *order=None*, *extract_orders=None*, *wunits=None*, ***kwargs*)

Bases: *jwst.datamodels.ReferenceFileModel*

A model for a reference file of type "wavelengthrange". The model is used by MIRI, NIRSPEC, NIRCAM, and NIRISS

Attributes Summary

reftype
schema_url

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
to_fits()	Write a DataModel to a FITS file.
validate()	Convenience function to be run when files are created.

Attributes Documentation

reftype = 'wavelengthrange'

schema_url = 'wavelengthrange.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

to_fits()
Write a DataModel to a FITS file.

Parameters

- **init** (*file path or file object*) –
- **kwargs** (*args,*) – Any additional arguments are passed along to `astropy.io.fits.writeto`.

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

WaveCorrModel

class jwst.datamodels.**WaveCorrModel**(*init=None*, *apertures=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

Attributes Summary

aperture_names
reftype
schema_url

Methods Summary

on_save([path])	This is a hook that is called just before saving the file.
populate_meta()	
validate()	Convenience function to be run when files are created.

Attributes Documentation

aperture_names

reftype = 'wavecorr'

schema_url = 'wavecorr.schema.yaml'

Methods Documentation

on_save(*path=None*)
This is a hook that is called just before saving the file. It can be used, for example, to update values in the metadata that are based on the content of the data.

Override it in the subclass to make it do something, but don't forget to "chain up" to the base class, since it does things there, too.

Parameters **path** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The path to the file that we're about to save to.

populate_meta()

validate()
Convenience function to be run when files are created. Checks that required reference file keywords are set.

WfssBkgModel

class jwst.datamodels.**WfssBkgModel**(*init=None*, ***kwargs*)
Bases: *jwst.datamodels.ReferenceFileModel*

A data model for 2D WFSS master background reference files.

Parameters

- **init** (*any*) – Any of the initializers supported by *DataModel*.
- **data** (*numpy array*) – The science data. 2-D.
- **dq** (*numpy array*) – The data quality array. 2-D.
- **err** (*numpy array*) – The error array. 2-D.
- **dq_def** (*numpy array*) – The data quality definitions table.

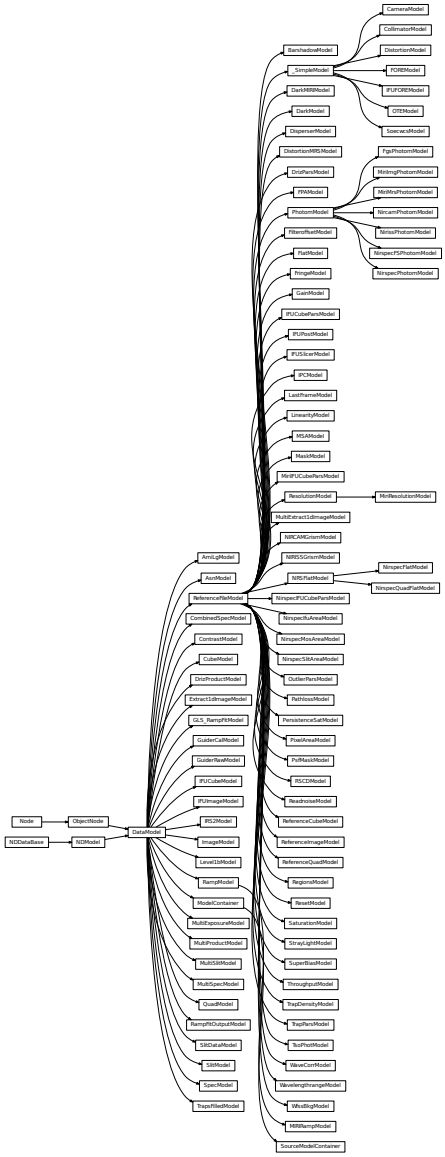
Attributes Summary

schema_url

Attributes Documentation

schema_url = 'wfssbkg.schema.yaml'

Class Inheritance Diagram



12.1.13 Data Quality (DQ) Initialization

Description

The Data Quality Initialization step in the calibration pipeline populates the Data Quality mask for the input dataset. DQ flags from the appropriate static mask reference file in CRDS are copied into the PIXELDQ object of the input dataset, because it is assumed that flags in the mask reference file pertain to problem conditions that are group- and

integration-independent.

The actual process consists of the following steps:

- Determine what mask reference file to use via the interface to the bestref utility in CRDS.
- If the PIXELDQ and GROUPDQ objects of the input dataset do not already exist, which is the case for raw Level-1b input products, create these objects in the input data model and initialize them to zero. The PIXELDQ array will be 2-D, with the same number of rows and columns as the input science data. The GROUPDQ array will be 4-D with the same dimensions (nints, ngroups, nrows, ncols) as the input science data array.
- 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.
- Copy the DQ flags from the reference file mask to the science data PIXELDQ array using numpy's bitwise_or function.

Step Arguments

The Data Quality Initialization step has no step-specific arguments.

Reference File Types

The Data Quality Initialization step uses a MASK reference file.

CRDS Selection Criteria

MASK reference files are currently selected based only on the value of DETECTOR in the input science data set. There is one MASK reference file for each JWST instrument detector.

MASK Reference File Format

The MASK reference file is a FITS file with a primary HDU, 1 IMAGE extension HDU and 1 BINTABLE extension. The primary data array is assumed to be empty. The MASK data are stored in the first IMAGE extension, which shall have EXTNAME='DQ'. The data array in this extension has integer data type and is 2-D, with dimensions equal to the number of columns and rows in a full frame raw readout for the given detector, including reference pixels. Note that this does not include the reference output for MIRI detectors.

The BINTABLE extension contains the bit assignments used in the DQ array. It uses `EXTNAME=DQ_DEF` and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

jwst.dq_init Package

Classes

DQInitStep([name, parent, config_file, ...])	DQInitStep: 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: *jwst.stpipe.Step*

DQInitStep: Initialize the Data Quality extension from the mask reference file. Also initialize the error extension

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.
- ****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

reference_file_types	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

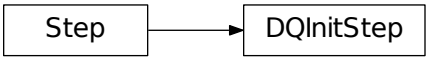
reference_file_types = ['mask']

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



12.1.14 Emission

Description

This step currently is a no-op; it passes the input file to the next step unchanged.

jwst.emission Package

Classes

EmissionStep([name, parent, config_file, ...])	EmissionStep: This step currently is a no-op; it passes the input file to the next step unchanged.

EmissionStep

class jwst.emission.**EmissionStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

EmissionStep: This step currently is a no-op; it passes the input file to the next step unchanged.

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.
- ****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.

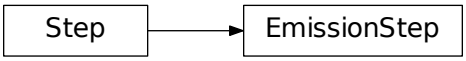
Methods Summary

process(input_file)	This is where real work happens.

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



12.1.15 Exposure to Source Conversion

Description

Overview

The `exp_to_source` is a Level2b->Level3 tool. It will take a list of Level2b MSA exposures and rearrange the data to produce files that are source-centric.

Usage

exp_to_source

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, ..]*) – List of MultiSlitModel instances to reformat.

Returns {**str** – Returns a dict of MultiExposureModel instances wherein each instance contains slits belonging to the same source. The key is the name of each source.

Return type *MultiExposureModel*, }

multislit_to_container

jwst.exp_to_source.**multislit_to_container**(*inputs*)
Reformat exposure-based MSA data to source-based containers.

Parameters **inputs** (*[MultiSlitModel, ..]*) – List of MultiSlitModel instances to reformat, or just a ModelContainer full of MultiSlitModels.

Returns {**str** – Returns a dict of ModelContainer instances wherein each instance contains ImageModels of slits belonging to the same source. The key is the name of each slit.

Return type *ModelContainer*, }

12.1.16 Extract 1D Spectra

Description

The extract_1d step extracts a 1-d signal from a 2-d or 3-d dataset and writes a spectrum to a product. This works on fixed-slit data (NIRSpec data through any one or more of the fixed slits, MIRI LRS data through the slit or in the slitless region, and NIRISS slitless data) as well as IFU data and NIRSpec MOS (micro-shutter array) data.

For GRISM data (NIS_WFSS or NRC_WFSS), no reference file is used. The extraction region is taken to be the full size of the input subarray or cutout, or it could be restricted to the region within which the world coordinate system is defined. The dispersion direction is the one along which the wavelengths change more rapidly.

For IFU data, the extraction options differ depending on whether the target is a point source or an extended source. For a point source, the spectrum will be extracted using circular aperture photometry, optionally including background subtraction using a circular annulus. For an extended source, rectangular aperture photometry will be used, with no background subtraction. The photometry makes use of astropy photutils. The region of overlap between an aperture and a pixel can be calculated by one of three different methods: "exact", limited only by finite precision arithmetic; "center", i.e. 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.

Input

Level 2-b countrate data, or level-3 data. The format should be a CubeModel, a SlitModel, an IFUCubeModel, an ImageModel, a DrizProductModel, a MultiSlitModel, a MultiProductModel, or a ModelContainer. The SCI extensions should have keyword SLTNAME to specify which slit was extracted, though if there is only one slit (e.g. full-frame data), the slit name can be taken from the JSON reference file instead.

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, ERROR, DQ, NET, NERROR, BACKGROUND, and BERROR. WAVELENGTH is the value calculated using the WCS. NET is the count rate minus background, in counts/pixel of spectral width, summed along the direction perpendicular to the dispersion. Currently only a simple summation is done, with no weighting. A more sophisticated algorithm will be introduced in future builds. BACKGROUND is the measured background, scaled to the extraction width used for the NET. BACKGROUND will

be zero if the reference file did not specify that background should be determined. FLUX will be computed from NET if there is a RELSENS table for the input slit; otherwise, FLUX will be zero. ERROR, DQ, NERROR, and BERROR are not populated with useful values yet.

Reference File

The reference file is a text file that uses JSON to hold the information needed.

CRDS Selection Criteria

The file is selected based on the values of DETECTOR and FILTER (and GRATING for NIRSpec).

Extract_1D Reference File 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 to use 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 (but for IFU data, see below). 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, that value will be used. If it was 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 source extraction region can be specified with `ystart`, `ystop`, etc., but a more flexible alternative is to use `src_coeff`. If background is to be subtracted, this should be specified by giving `bkg_coeff`. These are described in more detail below.

- 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)
- xstop: last pixel in the horizontal direction, X (int)
- ystart: first pixel in the vertical direction, Y (int)
- ystop: last pixel in the vertical direction, Y (int)
- 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_order: order of polynomial fit to background regions (int)
- extract_width: number of pixels in cross-dispersion direction (int)

If src_coeff is given, those coefficients take priority for specifying the source extraction region in the cross-dispersion direction. xstart and xstop (or ystart and ystop if dispaxis is 2) will still be used for the limits in the dispersion direction. Background subtraction will be done if and only if bkg_coeff is given. See below for further details.

For IFU cube data, these keys are used instead of the above:

- id: the slit name, but this can be "ANY" (string)
- x_center: X pixel coordinate of the target (pixels, float, the default is the center of the image along the X axis)
- y_center: Y pixel coordinate of the target (pixels, float, the default is the center of the image along the Y axis)
- radius: (only used for a point source) the radius of the circular extraction aperture (pixels, float, default is one quarter of the smaller of the image axis lengths)
- subtract_background: (only used for a point source) if true, subtract a background determined from an annulus with inner and outer radii given by inner_bkg and outer_bkg (boolean)
- inner_bkg: (only for a point source) radius of the inner edge of the background annulus (pixels, float, default = radius)
- outer_bkg: (only for a point source) radius of the outer edge of the background annulus (pixels, float, default = inner_bkg * sqrt(2))
- width: (only for an extended source) the width of the rectangular extraction region; if theta = 0, the width side is along the X axis (pixels, float, default is half of the smaller image axis length)
- height: (only for an extended source) the height of the rectangular extraction region; if theta = 0, the height side is along the Y axis (pixels, float, default is half of the smaller image axis length)
- angle: (only for an extended source) the counterclockwise rotation angle of the width side from the positive X axis (degrees)
- 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 5)

The rest of this description pertains to the parameters for non-IFU data.

If src_coeff is not given, the extraction limits can be specified by xstart, xstop, ystart, ystop, and extract_width. Note that all of these values are integers, and that the start and stop limits are inclusive. If dispaxis is 1, the zero-indexed limits in the dispersion direction are xstart and xstop; if dispaxis is 2, the dispersion limits are ystart and ystop. (The dispersion limits can be given even if src_coeff has been used for defining the cross-dispersion limits.) The limits in the cross-dispersion direction can be given by ystart and ystop (or xstart and xstop if dispaxis is 2). If extract_width is also given, that takes priority over ystart to ystop (for dispaxis = 1) for the extraction width, but ystart and ystop (for dispaxis = 1) will still be used to define the middle in the cross-dispersion direction. Any of these parameters can be modified by the step code if the extraction region would extend outside the input image, or outside the domain specified by the WCS.

The source extraction region can be specified more precisely by giving src_coeff, coefficients for polynomial functions for the lower and upper limits of the source extraction region. As described in the previous paragraph, using this key will override the values of ystart and ystop (if dispaxis is 1) or xstart and xstop (if dispaxis is 2), and extract_width. These polynomials are functions of either wavelength (in microns) or pixel number (pixels in the dispersion direction, with respect to the input 2-D slit image), specified by the key independent_var. The default is "pixel". The values of these polynomial functions are pixel numbers in the direction perpendicular to dispersion. More than one source extraction region may be specified, though this is not expected to be a typical case.

Background regions are specified by giving bkg_coeff, coefficients for polynomial functions for the lower and upper limits of one or more regions. Background subtraction will be done only if bkg_coeff is given in the reference

file. See below for an example. See also `bkg_order` below.

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

If the dispersion direction is vertical, replace "lower" with "left" and "upper" with "right" in the above description.

Note 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 (this should not be confused with `bkg_order`, described below). The number of coefficients must be at least one, and there is no predefined upper limit. 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 image (identified by key `id`).

The background is determined independently for each column (or row, if `dispaxis` is 2) of the spectrum. The `smoothing_length` parameter is the width of a boxcar for smoothing the background in the dispersion direction. If this is not specified, either in the reference file, the config file, or on the command line, no smoothing will be done along the dispersion direction. Following background smoothing (if any), for each column (row), a polynomial of order `bkg_order` will be fit to the pixel values in that column (row) in all the background regions. If not specified, a value of 0 will be used, i.e. a constant function, the mean value. The polynomial will then be 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.

Step Arguments

The extract_1d step has three step-specific arguments. Currently none of these is used for IFU data.

- `--smoothing_length`

If `smoothing_length` is greater than 1 (and is an odd integer), the background 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 reference file. If a value was specified in the reference file, that will be used. Note that in this case a different value can be specified for each slit. If no value was specified either by the user or in the reference file, no background smoothing will be done.

- `--bkg_order`

This is 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 reference file. If neither is specified, a value of 0 will be used.

- `--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".

jwst.extract_1d Package

Classes

Extract1dStep([name, parent, config_file, ...])	Extract1dStep: 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: *jwst.stpipe.Step*

Extract1dStep: Extract a 1-d spectrum from 2-d 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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

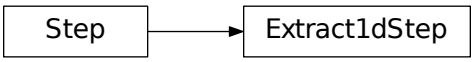
reference_file_types = ['extract1d']

spec = '\n # Boxcar smoothing width for background regions.\n smoothing_length = integ

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



12.1.17 Extract 2D Spectra

Description

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. It also computes an array of wavelengths. The SCI, ERR, DQ and WAVELENGTH arrays are stored as one or more `slit` objects in an output MultiSlitModel and saved as separate 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 WFSS modes in NIRCAM and NIRSS, no `bounding_box` has been attached to the datamodel. This is to keep the WCS flexible enough to be used with any source catalog 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 dispersed spectra for each object. One box is made for each order. `extract2d` uses the source catalog referenced in the input models 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.

Algorithm

The step is currently applied only to NIRSpec Fixed Slit, NIRSPEC MSA, NIRSPEC TSO, NIRCAM and NIRISS WFSS, and NIRCAM TSGRISM observations.

For NIRSPEC:

If the step parameter `slit_name` is left unspecified, the default behavior is to extract all slits which project on the detector. Only one slit may be extracted by specifying the slit name with the `slit_name` argument, using one of the following accepted names: `S1600A1`, `S200A1`, `S200A2`, `S200B1` or `S400A1` in the case of NIRSPEC FS exposure or any of the slitlet names in the case of the MSA.

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 MultiSlit 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`.

For NIRCAM WFSS and NIRISS WFSS :

If the step parameter `grism_objects` is left unspecified, the default behavior is to use the source catalog that is specified in the input model's meta information, `input_model.meta.source_catalog.filename`. Otherwise, a user can submit of list of `GrismObjects` that contains information about the objects that should be 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`.

For NIRCAM TSGRISM:

There is no source catalog created for TSO observations because the source is always placed on the same pixel, the user can only vary the size of the subarray. All of the subarrays have their "bottom" edge located at the physical bottom edge of the detector and grow in size vertically. The source spectrum trace will always be centered somewhere near row 34 in the vertical direction (dispersion running parallel to rows). So the larger subarrays will just result in larger amount of sky above the spectrum.

`extract_2d` will always produce a cutout that is 64 pixels in height (cross-dispersion direction) for all subarrays and full frame exposures, which 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.

Step Arguments

The `extract_2d` step has two optional arguments for NIRSPEC observations:

- `--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. Currently only used for NIRspec fixed slit exposures.

- `--apply_wavecorr`: bool (default is True). Flag indicating whether to apply the Nirspec wavelength zero-point correction.

For NIRCAM and NIRISS WFSS, the `extract_2d` step has three optional arguments:

- `--grism_objects`: list (default is empty). A list of `jwst.transforms.models.GrismObject`.
- `--mmag_extract`: float. (default is 99.) the minimum magnitude object to extract
- `--extract_orders`: list. The list of orders to extract. The default is taken from the `wavelengthrange` reference file.

For NIRCAM TSGRISM, the `extract_2d` step has two optional arguments:

- `--extract_orders`: list. The list of orders to extract. The default is taken from the `wavelengthrange` reference file.
- `--extract_height`: int. The cross-dispersion size to extract

Reference Files

To apply the Nirspec wavelength zero-point correction, this step uses the WAVECORR reference file. The zero-point correction is applied to observations with EXP_TYPE of "NRS_FIXEDSLT", "NRS_BRIGHTOBJ" or "NRS_MSASPEC". This is an optional correction (on by default). It can be turned off by specifying `apply_wavecorr=False` when running the step.

NIRCAM WFSS and NIRISS WFSS observations use the `wavelengthrange` reference file in order to construct the bounding boxes around each objects orders. If a list of ''GrismObject''s is supplied, then no reference file is neccessary.

For NIRCAM WFSS and TSGRIM modes and NIRISS WFSS mode the `wavelengthrange` file contains the wavelength limits to use when caluclating 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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types	
spec	

Methods Summary

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

Attributes Documentation

reference_file_types = ['wavecorr', 'wavelengthrange']

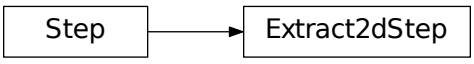
spec = '\n slit_name = string(default=None)\n apply_wavecorr = boolean(default=True)\n

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



12.1.18 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>
                <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 repesent 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'
```

(continues on next page)

(continued from previous page)

```
<<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')

<<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 contructed. 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
SUBYSIZE  = 0 / length of subarray along y-axis
LIGHTCOL  = 0 / Number of light-sensitive columns
```

(continues on next page)

(continued from previous page)

```
<<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

12.1.19 First Frame Correction

Description

The MIRI first frame correction step flags the first group in every integration as bad (the DO_NOT_USE group data quality flag is added), if the number of groups is greater than 1. No correction or flagging is done otherwise.

Reference File

This step does not use any reference file.

Step Arguments

The first frame correction has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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.

Methods Summary

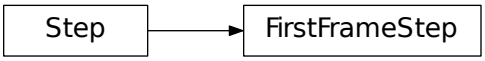
process(input)	This is where real work happens.

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



12.1.20 Flatfield

Description

At its basic level this step flat-fields an input science data set by dividing by a flat-field reference image. In particular, the SCI array from the flat-field reference file is divided into both the SCI and ERR arrays of the science data set, and the flat-field DQ array is combined with the science DQ array using a bit-wise OR operation.

Non-NIRSpec Data

MultiSlit data models are handled as follows. First, if the flat-field reference file supplied to the step is also in the form of a MultiSlit model, it searches the reference file for slits with names that match the slits in the science exposure (e.g. 'S1600A1' or 'S200B1'). When it finds a match, it uses the flat-field data for that slit to correct the particular slit data in the science exposure. If, on the other hand, the flat-field consists of a single image model, the region corresponding to each slit in the science data is extracted on-the-fly from the flat-field data and applied to the corresponding slit in the science data.

Multiple-integration datasets (the _rateints.fits products from the ramp_fit step) are handled by applying the flat-field to each integration.

NIRSpec imaging data are corrected the same as non-NIRSpec data, i.e. they will just be divided by a flat-field reference image.

For pixels whose DQ is NO_FLAT_FIELD in the reference file, the flat value is reset to 1.0. Similarly, for pixels whose flat value is NaN, the flat value is reset to 1.0 and DQ value in the output science data is set to NO_FLAT_FIELD. In both cases, the effect is that no flat-field is applied.

If any part of the input data model gets flat-fielded (e.g. at least one slit of a MultiSlit model), the status keyword S_FLAT will be set to COMPLETE in the output science data.

NIRSpec Data

Flat-fielding of NIRSpec spectrographic data differs from other modes in that the flat field array that will be divided into the SCI and ERR arrays of the input science data set 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 mode and on which slit or 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. See the Reference File section for further details. There is an option to save the on-the-fly flat field to a file.

NIRSpec NRS_BRIGHTOBJ data are processed much like other NIRSpec spectrographic data, except that NRS_BRIGHTOBJ data are in a CubeModel, rather than a MultiSlitModel or ImageModel (used for IFU data). A 2-D flat field image will be constructed on-the-fly as usual, but this image will be divided into each plane of the 3-D science data and error array, resulting in an output CubeModel.

When this step is called with NIRSpec imaging data as input, the data will be flat-fielded as described in the section for non-NIRSpec data.

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.

Reference File

There are four reference file types for the flat_field step. Reftype FLAT is used for all exposure types except NIRSpec spectra. NIRSpec spectra use three reftypes: FFLAT (fore optics), SFLAT (spectrograph optics), and DFLAT (detector).

CRDS Selection Criteria

For MIRI Imaging, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, FILTER, READPATT, and SUBARRAY in the science data file.

For MIRI MRS, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, BAND, READPATT, and SUBARRAY in the science data file.

For NIRCam, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, FILTER, and PUPIL in the science data file.

For NIRISS, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, and FILTER in the science data file.

For NIRSpec, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, FILTER, GRATING, and EXP_TYPE in the science data file.

Reference File Formats for MIRI, NIRCAM, and NIRISS

Except for NIRSpec modes, flat-field reference files are FITS format with 3 IMAGE extensions and 1 BINTABLE extension. The primary data array is assumed to be empty. The 3 IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	2	ncols x nrows	float
ERR	2	ncols x nrows	float
DQ	2	ncols x nrows	integer

The BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array.

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 1032 x 1024 for MIRI, unless data were taken in subarray mode.

For slit spectroscopy, a set of SCI, ERR and DQ extensions can be provided for each aperture (identified by the detector subarray onto which the spectrum is projected).

A single DQ_DEF extension provides the data-quality definitions for all of the DQ arrays, which must use the same coding scheme. The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Reference File Formats for NIRSpec

For NIRSpec 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 there may be 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, as the allocated array size can be larger than is 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 file 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, 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 flat field 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 flat field contains four sets (one for each MSA quadrant) of images, WAVELENGTH tables, and FAST_VARIATION tables. The images are unique to the fore optics flat fields, however. The image "pixels" correspond to micro-shutter array slits, rather than to detector pixels. The array size is 365 rows by 171 columns, and there are multiple planes to handle the slow variation of flat field with wavelength.

For the spectrograph optics, 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 flat field 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.

Fore Optics (FFLAT)

There are 3 types of FFLAT reference files: fixed slit, msa spec, and IFU. For each type the primary data array is assumed to be empty.

Fixed Slit

The fixed slit references 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

The MSA Spec references files have EXP_TYPE=NRS_MSASPEC, and contain data pertaining to each of the 4 quadrants. For each quadrant, there are 3 IMAGE extensions, a BINTABLE extension labeled WAVELENGTH, and a BINTABLE extension labeled FAST_VARIATION. The file also contains one BINTABLE labeled DQ_DEF.

The IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x nelem	float
ERR	3	ncols x nrows x nelem	float
DQ	3	ncols x nrows x nelem	integer

For all 3 of these extensions, the EXTVER keyword indicates the quadrant number, 1 to 4. Each plane of the SCI array gives the flat_field value for every pixel in the quadrant 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. There is a single row in this table, as the same wavelength-dependent value is applied to all pixels in the quadrant.

The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition

- DESCRIPTION: a string description of the condition

IFU

The IFU reference files have EXP_TYPE=NRS_IFU. These have one extensions, 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

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 determined by 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.

The DQ_DEF table contains the bit assignments used in the DQ arrays. The table contains the 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Spectrograph (SFLAT)

There are 3 types of SFLAT reference files: fixed slit, msa spec, and IFU. For each type the primary data array is assumed to be empty.

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 references files have EXP_TYPE=NRS_MSASPEC. There are 3 IMAGE extensions, a BINTABLE extension labeled WAVELENGTH, a BINTABLE extension labeled FAST_VARIATION, and a BINTABLE labeled DQ_DEF.

The IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x n_wl	float
ERR	3	ncols x nrows x n_wl	float
DQ	3	ncols x nrows x n_wl	integer

The keyword NAXIS3 in these 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 determined by 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.

The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Detector (DFLAT)

There is only one type of DFLAT reference file, and it contains 3 IMAGE extensions, a BINTABLE extension labeled WAVELENGTH, a BINTABLE extension labeled FAST_VARIATION, and a BINTABLE labeled DQ_DEF.

The IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x n_wl	float
ERR	3	ncols x nrows	float
DQ	3	ncols x nrows	integer

The keyword NAXIS3 in the SCI IMAGE 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, as the same wavelength-dependent value is applied to all pixels.

The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Step Arguments

The flat_field step has one step-specific argument, and it is only relevant for NIRSpec data.

- `--flat_suffix`

`flat_suffix` is a string, the suffix to use when constructing the name of on optional output file for on-the-fly flat fields. If `flat_suffix` is specified (and if the input data are NIRSpec), the extracted and interpolated flat fields will be saved to a file with this suffix. The default (if `flat_suffix` was not specified) is to not write this optional output file.

jwst.flatfield Package

Classes

FlatFieldStep([name, parent, config_file, ...])	FlatFieldStep: Flat-field a science image using a flat-field reference image.

FlatFieldStep

class jwst.flatfield.**FlatFieldStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

FlatFieldStep: Flat-field a science image using a flatfield 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.
- ****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

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

reference_file_types = ['flat', 'fflat', 'sflat', 'dflat']

spec = '\n # Suffix for optional output file for interpolated flat fields.\n # Note th

Methods Documentation

process(*input*)

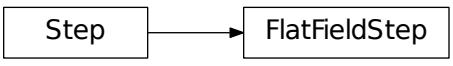
This is where real work happens. Every Step subclass has to override this method. The default behaviour is to raise a NotImplementedError exception.

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



12.1.21 Fringe Correction

Description

This step applies a fringe correction to the SCI data of an input data set by dividing the SCI and ERR arrays by a fringe reference image. In particular, the SCI array from the fringe reference file is divided into the SCI and ERR arrays of the science data set. Only pixels that have valid values in the SCI array of the reference file will be corrected.

This correction is applied only to MIRI MRS (IFU) mode exposures, which are always single full-frame 2-D images.

The input to this step is always 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 will be set to COMPLETE.

Reference File Types

The fringe correction step uses a FRINGE reference file, which has the same format as the FLAT reference file. This correction is applied only to MIRI MRS (IFU) mode exposures, which are always single full-frame 2-D images.

CRDS Selection Criteria

Fringe reference files are selected by DETECTOR and GRATNG14.

Reference File Format

Fringe reference files are FITS format with 3 IMAGE extensions and 1 BINTABLE extension. The primary data array is assumed to be empty. The 3 IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	2	ncols x nrows	float
ERR	2	ncols x nrows	float
DQ	2	ncols x nrows	integer

Image dimensions should be 1032 x 1024.

The BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

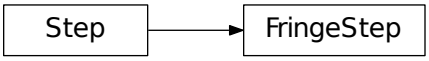
reference_file_types = ['fringe']

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



12.1.22 Gain Scale Processing

Description

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 both rescaled.

The scaling factor is obtained from the `GAINFACT` keyword in the header of the gain reference file. Normally the `ramp_fit` step will read that keyword value during its execution and store the value in the science data keyword `GAINFACT`, 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 will load the gain reference file to retreive it. If all attempts to find the scaling factor fail, the step will be 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 will cause 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 will be benign.

Upon successful completion of the step, the `S_GANSCL` keyword in the science data will be set to "COMPLETE."

Arguments

The gain_scale correction has no step-specific arguments.

Reference File

The gain_scale correction step uses the gain reference file. The only purpose of the reference file is to retrieve the `GAINFACT` keyword value from its header (the reference file data are not used in any way). If the `ramp_fit` step, which also uses the gain reference file, succeeded in finding the `GAINFACT` keyword in this reference file, it will store the value in the `GAINFACT` keyword in the science data, in which case the `gain_scale` step will not reload the gain reference file.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

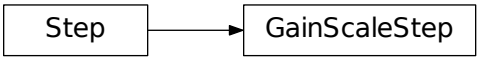
reference_file_types = ['gain']

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



12.1.23 Group Scale Processing

Description

The `group_scale` step rescales pixel values in raw JWST science data products in order to correct for the effect of using a value of NFRAMES for on-board frame averaging that is not a power of 2.

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 an 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 only results in the correct average 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 every group that are too low by the factor NFRAMES/FRMDIVSR.

This step rescales raw pixel values to the correct level by multiplying all groups in all integrations by the factor FRMDIVSR/NFRAMES.

It is assumed that this step will always be 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.

If the step detects that the values of NFRAMES and FRMDIVSR are equal to one another, which means the data were scaled correctly on-board, it skips processing and returns the input data unchanged. In this case, the calibration step status keyword `S_GRPSCL` will be set to `SKIPPED`. After successful correction of data that needs to be rescaled, the `S_GRPSCL` keyword will be set to `COMPLETE`.

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 NFRAMES that is a power of two.

GroupScaleStep

class jwst.group_scale.**GroupScaleStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

GroupScaleStep: Rescales group data to account for on-board frame averaging that did not use NFRAMES that is a power of two. 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.
- ****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.

Methods Summary

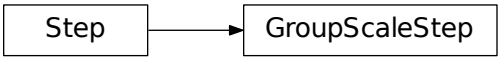
process(input)	This is where real work happens.

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



12.1.24 Guider CDS Processing

Description

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 has 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 will be 3D, with dimensions of (ncols x nrows x 1).

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 will be 3D, with dimensions of (ncols x nrows x nints).

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 will be 3D, with dimensions of (ncols x nrows x nints).

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 does not use any reference files.

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: *jwst.stpipe.Step*

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.
- ****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.

Methods Summary

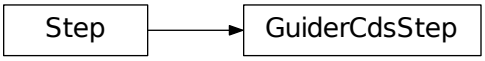
process(input)	This is where real work happens.

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



12.1.25 Imprint Subtraction

Description

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 the imprint exposure. Either of these arguments can be provided as either a file name or a JWST data model.

The SCI data array of the imprint exposure is subtracted from the SCI array of the target exposure. The DQ arrays of the two exposures are combined using a bit-wise logical OR operation. The ERR 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 exposure.

ImprintStep

class jwst.imprint.**ImprintStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

ImprintStep: Removes NIRSpec MSA imprint structure from an exposure by subtracting an imprint 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.
- ****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

spec

Methods Summary

process(input, imprint)	This is where real work happens.

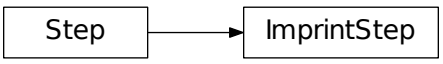
Attributes Documentation

spec = '\n '

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



12.1.26 IPC Correction

Description

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.

Reference File Types

The IPC deconvolution current step uses an IPC reference file.

CRDS Selection Criteria

IPC reference files are selected on the basis of INSTRUME and DETECTOR values for the input science data set.

IPC Reference File Format

IPC reference files are FITS files with one IMAGE extension, with EXTNAME value of 'SCI'. The FITS primary data array is assumed to be empty. The SCI extension contains a floating-point data array.

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, perhaps 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 point to point across the image. In this case, the axes that are most rapidly varying (the last two, in Python notation; the first two, in IRAF 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.

Step Arguments

The IPC deconvolution step has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

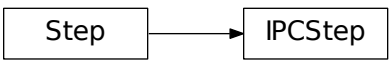
reference_file_types = ['ipc']

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



12.1.27 Jump Detection

Description

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 (bias drift) and non-linearity correction have been applied, so that the input data ramps do not have any non-linearities due to instrumental effects. The absence of any of these preceding corrections can lead to the false detection of jumps in the ramps, due to departure from linearity.

The step will automatically skip execution if the input data contain fewer than 3 groups per integration, because it's impossible to detect jumps with only 1 or 2 groups.

Algorithm

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 of the data is updated to reflect the location of each jump that was found, and the PIXELDQ array is updated to have DQ flags set to NO_GAIN_VALUE and DO_NOT_USE for all pixels that are non-positive or NaN in the gain 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 and Gordon, PASP 132, 1237 (2011).

Two-Point Difference Method

The two-point difference method is applied to each integration as follows:

- Compute the first differences for each pixel (the difference between adjacent groups)
- Compute the median of the first differences for each pixel
- 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 group
- Take the ratio of the first differences and the total noise for each group
- If the largest ratio is above the rejection threshold, flag the group corresponding to that ratio as having a jump
- If a jump is found, iterate the above steps with the jump-impacted group excluded, looking for additional jumps
- Stop iterating on a given pixel when no new jumps are found

Step Arguments

The Jump step has one optional argument that can be set by the user:

- `--rejection_threshold`: A floating-point value that sets the sigma threshold for jump detection.

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.

Reference File Types

The Jump step uses two reference files: GAIN and READNOISE. The gain values are used to temporarily convert the pixel values from units of DN to electrons. The read noise values are used as part of the noise estimate for each pixel. Both are necessary for proper computation of noise estimates.

CRDS Selection Criteria

GAIN Reference Files

The GAIN reference file is selected based on instrument, detector and, where necessary, subarray.

READNOISE Reference Files

The READNOISE reference file is selected by instrument, detector and, where necessary, subarray.

Reference File Formats

GAIN Reference Files

The gain reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of gain values (in e/DN) per pixel. The REFTYPE value is `GAIN`.

READNOISE Reference Files

The read noise reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of read noise values per pixel. The units of the read noise should be DN and should be the CDS (Correlated Double Sampling) read noise, i.e. the effective noise between any pair of non-destructive detector reads. The REFTYPE value is `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: *jwst.stpipe.Step*

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.
- ****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

do_yintercept	
reference_file_types	
spec	
yint_threshold	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

do_yintercept = False

reference_file_types = ['gain', 'readnoise']

spec = '\n rejection_threshold = float(default=4.0,min=0) # CR rejection threshold\n '

yint_threshold = 1.0

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

12.1.28 Last Frame Correction

Description

The last frame correction step flags the final group as bad (the GROUP data quality flags for the final group in all integrations are reset to DO_NOT_USE), if the number of groups is greater than 1. No correction or flagging is done otherwise.

Reference File

This step does not use any reference file.

Step Arguments

The last frame correction has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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.

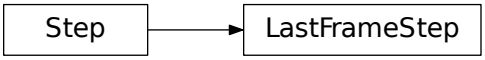
Methods Summary

process(input)	This is where real work happens.

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



12.1.29 Linearity Correction

Description

Assumptions

Beginning with the Build 5 pipeline, it is assumed that the input science exposure data from near-IR instruments have had the superbias subtraction applied, therefore the correction coefficients stored in the linearity reference files for those instruments must also have been derived from data that had the zero group subtracted.

It is also assumed that the saturation step has already been applied to the science data, so that saturation flags are set in the GROUPDQ array of the input science data.

Algorithm

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. Pixels having at least one correction coefficient equal to NaN (not

a number), or are flagged with "Linearity Correction not determined for pixel" (NO_LIN_CORR) in the PIXELDQ array will not have the linearity correction applied. Pixel values flagged as saturated in the GROUPDQ array for a given group will not have the linearity correction applied. All non-saturated groups for such a pixel will have the correction applied.

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 $F_c = c_0 + c_1F + c_2F^2 + c_3F^3...$

where F is the observed counts (in DN), c_n are the polynomial coefficients, and F_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.

The ERR array of the input science exposure is not modified.

The values from the linearity reference file DQ array are propagated into the PIXELDQ array of the input science exposure using a bitwise OR operation.

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 necessary.

Reference File Types

The linearity correction step uses a LINEARITY reference file.

CRDS Selection Criteria

Linearity reference files are selected by INSTRUME and DETECTOR.

Reference File Format

Linearity reference files are FITS format with 2 IMAGE extensions and 1 BINTABLE extension. The primary data array is assumed to be empty. The 2 IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
COEFFS	3	ncols x nrows x ncoeffs	float
DQ	2	ncols x nrows	integer

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 BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array.

Arguments

The linearity correction has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types	

Methods Summary

process(input)	This is where real work happens.

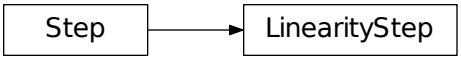
Attributes Documentation

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



12.1.30 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* based on some model defined in *DataModels*. 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 ouput 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, verbose])	Run main interface for blending metatdata 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*, *verbose=False*)

Run main interface for blending metatdata 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.
- **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 jwst.model_blender.blender import blendmodels
>>> from jwst import datamodels
>>> asnfile = "jw99999-a3001_20170327t121212_coron3_001_asn.json"
>>> asn = datamodels.open(asnfile)
>>> input_models = [asn[3],asn[4]]  # we know the last datasets are SCIENCE
>>> blendmodels(asn.meta.resample.output, inputs=input_models)
```

Alternatively, the filenames for all the inputs could be provided directly instead using:

```
>>> from jwst.associations import load_asn
>>> asn = load_asn(open(asnfile))
>>> input_names = [i['expname'] for i in asn['products'][0]['members'][3:]]
>>> blendmodels(asn['products'][0]['name'], inputs=input_names)
```

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.DataModel 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 instancs, 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 instancs, according to the given specification.

Parameters:

- *input_models* is a sequence where each element is either:
  - a datamodels.DataModel 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://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html#numpy.mean)
  * median: `numpy.median` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.median.html#numpy.med
  * maximum: `numpy.max`
  * minimum: `numpy.min`
  * sum: `numpy.sum` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html#numpy.sum)
  * standard deviation: `numpy.std` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html#numpy.st

  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}
```

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
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

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.
OrderedDict	Dictionary that remembers insertion order

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 ..
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
```

(continues on next page)

(continued from previous page)

```
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'.

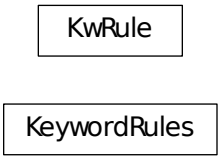
Methods Summary

interpret(hdr)	Use metadata to interpret rule.

Methods Documentation

interpret(*hdr*)
Use metadata to interpret rule.

Class Inheritance Diagram



jwst.model_blender Package

12.1.31 MIRI MRS Sky Matching

Description

Overview

The `mrs_imatch` step that "matches" image intensities of several input 2D MIRI MRS images by fitting polynomials to cube intensities (cubes built from input 2D images) in such a way as to minimize inter-image mismatch in the least squares sense. These "background matching" polynomials are defined in terms of world coordinates (e.g., `RA`, `DEC`, `lambda`).

Assumptions

Because polynomials are defined in terms of world coordinates, and because the algorithm needs to build 3D cubes for each input image, input images need to have valid WCS.

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 intensities of 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 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 argument:

- `bkg_degree`: An integer background polynomial degree (Default: 1)
- `subtract`: A boolean value indicating whether the computed matching "backgrounds" should be subtracted from image data (Default: `False` (https://docs.python.org/3/library/constants.html#False)).

Reference Files

This step does not require any reference files.

Also See

See *wiimatch package documentation* for more details.

LSQ Equation Construction and Solving

JWST pipeline step for image intensity matching for MIRI images.

Authors Mihai Cara

class jwst.mrs_imatch.mrs_imatch_step.**MRSIMatchStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

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.
- ****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.

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.

reference_file_types = []

spec = "\n # General sky matching parameters:\n bkg_degree = integer(min=0, default=1)

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.ImageModel*) – A *jwst.datamodels.ImageModel* 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* (https://docs.python.org/3/library/constants.html#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.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(images)	This is where real work happens.

Attributes Documentation

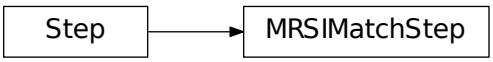
reference_file_types = []

spec = "\n # General sky matching parameters:\n bkg_degree = integer(min=0, default=1)

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



12.1.32 MSAFlagOpen Correction

Description

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 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 FAILEDOPENFLAG.

Reference File

The msaflagopen correction step uses a MSAOPER reference file.

CRDS Selection Criteria

NIRSPEC USEAFTER

Msaoper reference files are selected on the basis of USEAFTER date only. They are valid for NIRSpec only.

MSAOPER Reference File Format

The MSAOPER reference files are json files.

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

Step Arguments

The msaflagopen correction has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

reference_file_types = ['msaoper']

spec = '\n\n '

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

12.1.33 Outlier Detection

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 ambiguous for detection during ramp fitting. The outlier detection step implements the following algorithm to identify and flag any remaining cosmic-rays or other artifacts left over from previous calibrations:

- 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
- 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)
- create "blotted" data from the median image to exactly match each original input dataset
- 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
- 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

- Imaging data
- IFU spectroscopic data
- TSO data
- coronagraphic data
- spectroscopic data

This allows the outlier_detection step to be tuned to the variations in each type of JWST data.

Outlier Detection Code API

Python Step Design: OutlierDetectionStep

This module provides the sole interface to all methods of performing outlier detection on JWST observations. The `OutlierDetectionStep` supports multiple algorithms and determines the appropriate algorithm for the type of observation being processed. This step supports:

- **Image modes**: 'NRC_IMAGE', 'MIR_IMAGE', 'NIS_IMAGE', 'FGS_IMAGE'
- **Spectroscopic modes**: 'NRC_WFSS', 'MIR_LRS-FIXEDSLIT', 'NRS_FIXEDSLIT', 'NRS_MSASPEC', 'NIS_WFSS'
- **Time-Series-Observation(TSO) Spectroscopic modes**: 'NIS_SOSS', 'MIR_LRS-SLITLESS', 'NRC_TSGRISM', 'NRS_BRIGHTOBJ'
- **IFU Spectroscopic modes**: 'NRS_IFU', 'MIR_MRS'
- **TSO Image modes**:'NRC_TSIMAGE'
- **Coronagraphic Image modes**: 'NRC_CORON', 'MIR_LYOT', 'MIR_4QPM'

This step uses the following logic to apply the appropriate algorithm to the input data:

- Interpret inputs (ASN table, ModelContainer or CubeModel) to identify all input observations to be processed
- Read in type of exposures in input by interpreting `meta.exposure.type` from inputs
- Read in parameters set by user.
- Select outlier detection algorithm based on exposure type
  - **Images**: like those taken with NIRCam, will use *`OutlierDetection`* as described in *Default OutlierDetection Algorithm*
  - **Coronagraphic observations**: use *`OutlierDetection`* with resampling turned off as described in *Default OutlierDetection Algorithm*
  - **Time-Series Observations(TSO)**: both imaging and spectroscopic modes, will use *`OutlierDetection`* with resampling turned off as described in *Default OutlierDetection Algorithm*
  - **NIRSpec and MIRI IFU observations**: use *`OutlierDetectionIFU`* as described in *OutlierDetection for IFU Data*
  - **Long-slit spectroscopic observations**: use *`OutlierDetectionSpec`* as described in *OutlierDetection for Long-Slit Spectroscopic Data*
- Instantiate and run outlier detection class determined for the exposure type using parameter values interpreted from inputs.
- 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: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

check_input()	Use this method to determine whether input is valid or not.
process(input)	Perform outlier detection processing on input data.

Attributes Documentation

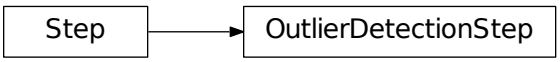
spec = "\n weight_type = option('exptime','error',None,default='exptime')\n pixfrac = :

Methods Documentation

check_input()
Use this method to determine whether input is valid or not.

process(*input*)
Perform outlier detection processing on input data.

Class Inheritance Diagram



Default OutlierDetection Algorithm

This module serves as the interface for applying outlier_detection to direct image observations, like those taken with NIRCam. The code implements the basic outlier detection algorithm used with HST data, as adapted to JWST.

Specifically, this routine performs the following operations:

- Extract parameter settings from input model and merge them with any user-provided values. The full set of user parameters includes:

```
wht_type: type of data weighting to use during resampling;
          options are 'exptime','error','None' [default='exptime']
pixfrac: pixel fraction used during resampling;
         valid values go from 0.0-1.0 [default=1.0]
kernel: name of resampling kernel; options are 'square', 'turbo', 'point',
        'lanczos', 'tophat' [default='square']
fillval: value to use to replace missing data when resampling;
          any floating point value (as a string) is valid (default='INDEF')
nlow: Number (as an integer) of low values in each pixel stack to ignore
      when computing median value [default=0]
nhigh:  Number (as an integer) of high values in each pixel stack to ignore
      when computing median value [default=0]
maskpt: Percent of maximum weight to use as lower-limit for valid data;
        valid values go from 0.0-1.0 [default=0.7]
grow: Radius (in pixels) from bad-pixel for neighbor rejection [default=1]
snr: Signal-to-noise values to use for bad-pixel identification; valid
     values are a pair of floating-point values in a single string
     [default='4.0 3.0']
scale: Scaling factor applied to derivative used to identify bad-pixels;
       valid value is a string with 2 floating point values [default='0.5 0.4')]
```

(continues on next page)

(continued from previous page)

```
backg: user-specified background value to apply to median image;
       [default=0.0]
save_intermediate_results: specifies whether or not to save any products
                           created during outlier_detection [default=False]
resample_data: specifies whether or not to resample the input data [default=True]
good_bits: List of DQ integer values which should be considered good when
           creating weight and median images [default=0]
```

- 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*, convert it into a ModelContainer. This allows each plane of the cube to be treated as a separate 2D image for resampling (if done at all) and for combining into a median image.
- By default, resample all input images into grouped observation mosaics; for example, combining all NIRCam multiple detector images from a single exposure or from a dithered set of exposures. (https://jwst-docs.stsci.edu/display/JTI/NIRCam+Dithers+and+Mosaics)
  - Resampled images will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
  - **If resampling was turned off**, a copy of the input (as a ModelContainer) will be used for subsequent processing.
- Create a median image from all grouped observation mosaics.
  - The median image will be created by combining all grouped mosaic images or non-resampled input data (as planes in a ModelContainer) pixel-by-pixel.
  - Median image will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True).
- By default, the median image will be blotted back (inverse of resampling) to match each original input exposure.
  - Resampled/blotted images will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
  - **If resampling was turned off**, the median image will be compared directly to each input image.
- Perform statistical comparison between blotted image and original image to identify outliers.
- Update input data model DQ arrays with mask of detected outliers.

Outlier Detection for TSO data

Time-series observations (TSO) data results in input data stored as a CubeModel where each plane in the cube represents a separate readout without changing the pointing. Normal imaging data would benefit from combining all readouts into a single, however, TSO data's value comes from looking for variations from one readout to the next. The outlier_detection algorithm, therefore, gets run with a few variations to accomodate the nature of the data.

- Input data is converted from a CubeModel (3D data array) to a ModelContainer
  - Each model in the ModelContainer is a separate plane from the input CubeModel
- The median image is created without resampling the input data
  - All readouts are aligned already, so no resampling needs to be performed

- A matched median gets created by combining the single median frame with the noise model for each input readout.
- Perform statistical comparison between the matched median with each input readout.
- 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.

Outlier Detection for IFU data

Integral-field unit (IFU) data gets readout as a 2D array. This 2D image then gets converted into a properly calibrated spectral cube (3D array) and stored as an IFUCubeModel for outlier detection. The many differences in data format for the IFU data relative to normal direct imaging data requires special processing in order to perform outlier detection in the IFU data.

- Convert the input IFUImageModel into a CubeModel using *CubeBuildStep*
  - A separate CubeModel will be generated for each channel using the `single` option for the *CubeBuildStep*.
- All input CubeModels then get median combined to create a single median IFUCubeModel product.
- The IFUCubeModel median product then gets resampled back to match each original input IFUImageModel dataset.
  - This resampling uses `CubeBlot` to perform this conversion.
- The blotted, median data then gets compared statistically to the original input data to detect outliers.
- The DQ array of each input dataset then gets updated to document the detected outliers.

jwst.outlier_detection.outlier_detection Module

Primary code for performing outlier detection on JWST observations.

Functions

flag_cr(sci_image, blot_image, **pars)	Masks outliers in science image.
abs_deriv(array)	Take the absolute derivate of a numpy array.

flag_cr

jwst.outlier_detection.outlier_detection.**flag_cr**(*sci_image*, *blot_image*, ***pars*)

Masks outliers in science image.

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) – the science data
- **blot_image** (ImageModel) – the blotted median image of the dithered science frames

- **pars** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – the user parameters for Outlier Detection
- **parameters** (*Default*) –
- **= 1 # Radius to mask [default=1 for 3x3]** (*grow*) –
- **= 0 # Length of CTE correction to be applied** (*ctegrow*) –
- **= "5.0 4.0" # Signal-to-noise ratio** (*snr*) –
- **= "1.2 0.7" # scaling factor applied to the derivative** (*scale*) –
- **= 0 # Background value** (*backg*) –

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 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 - does not use buffers to try to minimize memory usage - `astropy.stats.sigma_clipped_stats` replaces `stsci.imagestats.ImageStats` - `stsci.image.median` replaces `stsci.image.numcombine.numCombine`

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 (https://docs.python.org/3/library/constants.html#None)

do_detection()
Flag outlier pixels in DQ of input images.

Class Inheritance Diagram

OutlierDetection 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 used with HST data, as adapted to JWST IFU observations.

Specifically, this routine performs the following operations (modified from *Default Outlier Detection Algorithm*):

- Extract parameter settings from input model and merge them with any user-provided values
  - the same set of parameters available to *Default Outlier Detection Algorithm* also applies to this code
- Resample all input *IFUImageModel* images into *IFUCubeModel* observations.
  - Resampling uses *CubeBuildStep* to create *IFUCubeModel* formatted data for processing.
  - Resampled cubes will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
- Creates a median image from the set of resampled *IFUCubeModel* observations
  - Median image will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
- Blot median image to match each original input exposure.
  - Resampled/blotted cubes will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
- Perform statistical comparison between blotted image and original image to identify outliers.
- Updates input data model DQ arrays with mask of detected outliers.

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: *jwst.outlier_detection.outlier_detection.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 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 ModelContainer and merges
   them with any user-provided values
2. Resamples all input images into IFUCubeModel observations.
3. Creates a median image from all IFUCubeModels.
4. Blot median image using CubeBlot to match
    each original input ImageModel.
5. Perform statistical comparison between blotted image and original
   image to identify outliers.
6. 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
- **drizzled_models** (*list of objects*) – ModelContainer containing drizzled grouped input images
- **reffiles** (dict of *jwst.datamodels.DataModel*) – Dictionary of datamodels. Keys are reffile_types.

Attributes Summary

default_suffix

Methods Summary

blot_median(median_image)	IFU-specific version of blot_median.
create_median(resampled_models)	IFU-specific version of create_median.
do_detection()	Flag outlier pixels in DQ of input images.

Attributes Documentation

default_suffix = 's3d'

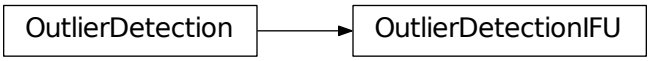
Methods Documentation

blot_median(*median_image*)
IFU-specific version of blot_median.

create_median(*resampled_models*)
IFU-specific version of create_median.

do_detection()
Flag outlier pixels in DQ of input images.

Class Inheritance Diagram



OutlierDetection for Long-Slit Spectroscopic Data

This module serves as the interface for applying outlier_detection to long-slit 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*):

- Extract parameter settings from input model and merge them with any user-provided values
  - the same set of parameters available to *Default Outlier Detection Algorithm* also applies to this code
- 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*, convert it into a *ModelContainer*. This allows each plane of the cube to be treated as a separate 2D image for resampling (if done at all) and for combining into a median image.

- Resamples all input images into a *ModelContainer* using `ResampleSpecData`
  - Resampled images will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
  - **If resampling was turned off**, the original inputs will be used to create the median image for cosmic-ray detection.
- Creates a median image from (possibly) resampled *ModelContainer*
  - Median image will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
- Blot median image to match each original input exposure.
  - Resampled/blotted images will be written out to disk if `save_intermediate_results` parameter has been set to `True` (https://docs.python.org/3/library/constants.html#True)
  - **If resampling was turned off**, the median image will be used as for comparison with the original input models for detecting cosmic-rays.
- Perform statistical comparison between blotted image and original image to identify outliers.
- Updates 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: *jwst.outlier_detection.outlier_detection.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 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 *jwst.datamodels.DataModel*) – 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

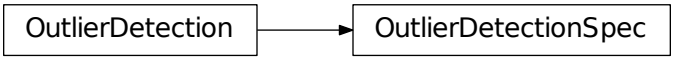
default_suffix = 's2d'

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: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

check_input()	Use this method to determine whether input is valid or not.
process(input)	Perform outlier detection processing on input data.

Attributes Documentation

spec = "\n weight_type = option('exptime','error',None,default='exptime')\n pixfrac = :

Methods Documentation

check_input()
Use this method to determine whether input is valid or not.

process(*input*)
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: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

process(input)	Step interface to running outlier_detection.

Attributes Documentation

spec = "\n weight_type = option('exptime','error',None,default='exptime')\n pixfrac = :

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: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

process(input)	Step interface for performing outlier_detection processing.

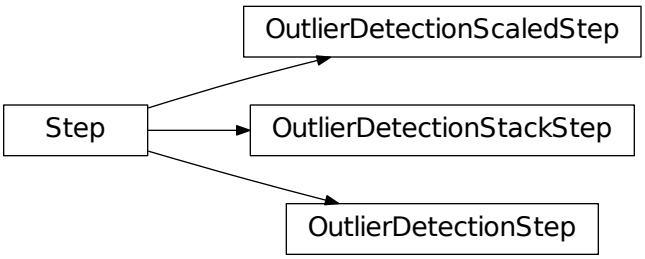
Attributes Documentation

spec = "\n weight_type = option('exptime','error',None,default='exptime')\n pixfrac = :

Methods Documentation

process (*input*)
Step interface for performing outlier_detection processing.

Class Inheritance Diagram



12.1.34 Pathloss Correction

Description

Overview

The pathloss correction step calculates the correction to apply to spectra when the 1-d extraction is performed. The motivation behind the correction is different for NIRSPEC and NIRISS, while that for MIRI has not been implemented yet. For NIRSPEC, this correction accounts for losses in the optical system due to light being scattered outside the grating, and to light not passing through the aperture, while for NIRISS SOSS data it corrects for the flux that falls outside the subarray.

Background

The correction is applicable to NIRSPEC IFU, MSA and FIXEDSLIT exposure types, to NIRISS SOSS data, and to MIRI LRS and MRS data, although the MIRI correction has not been implemented yet. 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 the pathloss 1-d array as a function of wavelength by interpolating in the pathloss cube at the position of the 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 array calculated by interpolating each plane of the pathloss cube at the position of the source (which is taken from datamodel). There are pairs of these arrays for both pointsource and uniformsource data types.

For the uniform source pathloss calculation, there is no dependence on position in the aperture, so the array of pathlosses and calculated wavelengths are attached to the datamodel.

Using 1-d arrays for the pathloss is different from what is suggested in the Ferruit document, where it is recommended that 2-d arrays of pathloss correction are attached to the data. However, since the only variable in the 2-d array is the wavelength, it was decided to simplify the process (and remove the possibility of incorrect usage) by creating 1-d arrays of pathloss and wavelength, which are to be applied at the time of 1-d extraction.

2-d arrays of the pathloss correction are also provided.

NIRISS

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 simply calculates the correction for each column by interpolating along the Pupil Wheel position dimension of the reference file using linear interpolation. The 1-d vector of correction vs. column number is attached to the science data in the PATHLOSS_POINTSOURCE extension, and can be obtained from the ImageModel using the .pathloss_pointsource attribute. This is a vector of length 2048 which gives the correction to be applied to each column of the science data, in the sense that the correction should be divided into the data to correct it.

Reference File

The pathloss correction step uses a pathloss reference file.

CRDS Selection Criteria

Pathloss reference files are selected on the basis of EXP_TYPE values for the input science data set. Only NIRSPEC IFU, FIXEDSLIT and MSA data, and NIRISS SOSS data perform a pathloss correction.

Pathloss Reference File Format

The PATHLOSS reference files are FITS files with extensions for each of the aperture types. The FITS primary data array is assumed to be empty.

The NIRSPEC IFU reference file just has 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 NIRSPEC FIXEDSLIT reference file has this FITS structure:

No.	Name	Type	Dimensions	Format
0	PRIMARY	PrimaryHDU	()	
1	PS	ImageHDU	(21, 21, 21)	float64
2	PSVAR	ImageHDU	(21, 21, 21)	float64
3	UNI	ImageHDU	(21,)	float64
4	UNIVAR	ImageHDU	(21,)	float64
5	PS	ImageHDU	(21, 21, 21)	float64
6	PSVAR	ImageHDU	(21, 21, 21)	float64
7	UNI	ImageHDU	(21,)	float64
8	UNIVAR	ImageHDU	(21,)	float64
9	PS	ImageHDU	(21, 21, 21)	float64
10	PSVAR	ImageHDU	(21, 21, 21)	float64
11	UNI	ImageHDU	(21,)	float64
12	UNIVAR	ImageHDU	(21,)	float64
13	PS	ImageHDU	(21, 21, 21)	float64
14	PSVAR	ImageHDU	(21, 21, 21)	float64
15	UNI	ImageHDU	(21,)	float64
16	UNIVAR	ImageHDU	(21,)	float64

HDU #1-4 are for the S200A1 aperture, while #5-8 are for S200A2, #9-12 are for S200B1 and #13-16 are for S1600A1. Currently there is no information for the S400A1 aperture.

The NIRSPEC IFU reference file just has 4 extensions after the primary HDU, as the behavious or each slice is considered identical.

The NIRSPEC MSASPEC reference file has 2 sets of 4 extensions, one for the 1x1 aperture size, and one for the 1x3 aperture size. Currently there are no other aperture sizes.

The NIRISS SOSS reference file has 1 extension in addition to the primary header unit. It contains a 3-dimensional array of float32 correction values. The dimensions of the array are 1x2040x17. The first dimension 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).

The headers associated with the PS extensions should contain the WCS information that describes what variables the correction depends on and how they relate to the dimensions of the correction array.

For the NIRSPEC reference files (MSASPEC, FIXEDSLIT and IFU), the WCS keywords should look like this:

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

This dimension expresses the decenter along the dispersion direction 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

This dimension expresses the decenter along the direction perpendicular to the dispersion for a point source

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 (should be 'WAVELENGTH')

This dimension expresses the change of correction with wavelength

The NIRISS SOSS reference file should also have WCS components, but their interpretation is different from those in the NIRSPEC reference file:

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

This dimension expresses the decenter along the dispersion direction for a point source

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

This dimension expresses the decenter along the direction perpendicular to the dispersion for a point source

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 (should be 'WAVELENGTH')

This dimension expresses the change of correction with wavelength

Step Arguments

The pathloss correction has no step-specific arguments.

jwst.pathloss Package

Classes

PathLossStep([name, parent, config_file, ...])	PathLossStep: Inserts the pathloss and wavelength arrays into the data.

PathLossStep

class jwst.pathloss.**PathLossStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

PathLossStep: Inserts the pathloss and wavelength arrays into the data.

Pathloss depends on the centering of the source in the aperture if the source is a point source. This step fills the following attributes in the datamodel:

- for exposure type NRS_IFU, the 1-d arrays .wavelength_pointsource, .pathloss_pointsource, .wavelength_uniformsource and .pathloss_uniformsource
- for exposure types NRS_FIXEDSLIT, NRS_BRIGHTOBJ, and NRS_MSASPEC, the 1-d arrays .slits[n].wavelength_pointsource, .slits[n].pathloss_pointsource, .slits[n].wavelength_uniformsource and .slits[n].pathloss_uniformsource

In all of these EXP_TYPES, these arrays are added to each member of the slits attribute.

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.
- ****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

reference_file_types	
spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

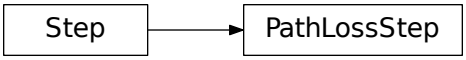
reference_file_types = ['pathloss']

spec = '\n '

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



12.1.35 Persistence

Description

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.

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.

Since 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:

$$\begin{aligned} trapsfilled = 2 \cdot (&trapdensity \cdot slope^2 \\ &\cdot (\Delta t^2 \cdot (par0 + par2)/2 + par0 \cdot (\Delta t \cdot \tau + \tau^2) \\ &\cdot exp(-\Delta t/\tau) - par0 \cdot \tau^2)) \end{aligned}$$

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 specified `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.

Reference File

There are three reference file types for the persistence step: TRAPDENSITY, PERSAT, and TRAPPARS.

CRDS Selection Criteria

Persistence reference files are selected by INSTRUME and DETECTOR.

At the present time, there are no reference files for MIRI, and CRDS will return "N/A" for the names of the files if the persistence step is run on MIRI 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'.

Reference File Formats

The TRAPDENSITY reference file contains an IMAGE extension that gives the density of traps at each pixel.

The PERSAT reference file contains an IMAGE extension that gives the persistence saturation threshold (full well) at each pixel.

The TRAPPARS reference file contains a BINTABLE extension with four float (double precision) columns:

- capture0: the coefficient of the exponential capture term
- capture1: minus the reciprocal of the capture e-folding time
- capture2: the "instantaneous" capture coefficient
- decay_param: minus the reciprocal of the decay e-folding time

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 have received a persistence correction greater than or equal to `flag_pers_cutoff` DN (the default is 40) will be flagged in the pixeldq extension of the output file.

- `--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".

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

reference_file_types = ['trapdensity', 'trappars', 'persat']

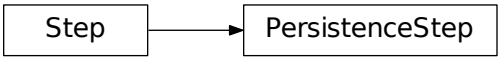
spec = '\n # `input_trapsfilled` is the name of the most recent trapsfilled\n # file f

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



12.1.36 Photometric Correction

Description

The photom step loads - and in some cases applies - information into a data product that allows for the conversion of count rates to absolute flux units. The flux conversion information is read from the photometric reference file. The exact nature of the information that's stored in the reference file and loaded into the science data product depends on the instrument mode.

Upon successful completion of this step, the status keyword S_PHOTOM will be set to COMPLETE.

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 copies the calibration information from that table row into the science product. 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 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). The scalar conversion constant in a selected table row is copied into the keyword PHOTMJSR in the primary header of the science product. The value of PHOTMJSR can then be used to convert data from units of DN/sec to MJy/steradian. The step also computes, on the fly, the equivalent conversion factor for converting the data to units of microJy/square-arcsecond and stores this value in the header keyword PHOTUJA2.

If the photom step finds that the wavelength and relative response arrays are populated in the selected table row, it copies those arrays to a table extension called "RELSENS" in the science data product.

None of the conversion factors are actually applied to the data for these observing modes. They are simply attached to the science product.

Pixel Area Data

For imaging modes, the photom step loads data from a pixel area map reference file and appends it to the science data product. The 2D data array from the pixel area map is copied into an image extension called "AREA" in the science data product.

The process of attaching the pixel area data also populates the keywords PIXAR_SR and PIXAR_A2 in the primary header of the science product, which give the average pixel area in units of steradians and square arcseconds, respectively. Both the photom and pixel area reference files contain the average pixel area values in their primary headers. The photom step copies the values from the pixel area reference file to populate the PIXAR_SR and PIXAR_A2 keywords in the science data. It will issue a warning if the values of those keywords in the two reference files differ by more than 0.1%.

NIRSpec IFU

The photom step uses the same type of tabular 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 retreives the scalar conversion constant, as well as the 1D wavelength and relative response arrays, from that row. It also loads the IFU pixel area data from the pixel area reference file.

It then uses the scalar conversion constant, the 1D wavelength and relative response, and pixel area data to compute a 2D sensitivity map (pixel-by-pixel) for the entire 2D science image. The 2D SCI and ERR arrays in the science exposure are divided by the 2D sensitivity map, which converts the science pixels from units of DN/sec to mJy/arcsec^2. Furthermore, the 2D sensitivity array is stored in a new extension of the science exposure called "RELSENS2D". The BUNIT keyword value in the SCI and ERR extension headers of the science product are updated to reflect the change in units.

MIRI MRS

For the MIRI MRS mode, the photom reference file contains 2D 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 2D sensitivity map (pixel-by-pixel) for the entire science image. This is divided into both the SCI and ERR arrays of the science exposure, which converts the pixel values from units of DN/sec to mJy/arcsec^2. The 2D sensitivity array is also stored in a "RELSENS2D" extension of the science exposure. The BUNIT keyword value in the SCI and ERR extension headers of the science product are updated to reflect the change in units.

Reference Files

The photom step uses a photom reference file and a pixel area map reference file. The pixel area map reference file is only used when processing imaging and NIRSpec IFU observations.

CRDS Selection Criteria

PHOTOM Reference Files

For FGS, photom reference files are selected based on the values of DETECTOR in the science data file.

For MIRI photom reference files are selected based on the values of DETECTOR and BAND in the science data file.

For NIRCam, photom reference files are selected based on the values DETECTOR in the science data file.

For NIRISS, photom reference files are selected based on the values of DETECTOR in the science data file.

For NIRSpec, photom reference files are selected based on the values of EXP_TYPE in the science data file.

A row of data within the table that matches the mode of the science exposure is selected by the photom step based on criteria that are instrument mode dependent. The current row selection criteria are:

- FGS: No selection criteria (table contains a single row)

- **MIRI:**
  - Imager (includes LRS): Filter and Subarray
  - MRS: Does not use table-based reference file
- NIRCam: Filter and Pupil
- **NIRISS:**
  - Imaging: Filter and Pupil
  - Spectroscopic: Filter, Pupil, and Order number
- **NIRSpec:**
  - Fixed Slits: Filter, Grating, and Slit name
  - IFU and MOS: Filter and Grating

AREA map Reference Files

For FGS, photom reference files are selected based on the values of DETECTOR in the science data file.

For MIRI photom reference files are selected based on the values of DETECTOR and EXP_TYPE in the science data file.

For NIRCam, photom reference files are selected based on the values of DETECTOR and EXP_TYPE in the science data file.

For NIRISS, photom reference files are selected based on the values of DETECTOR and EXP_TYPE in the science data file.

For NIRSpec, photom reference files are selected based on the values of DETECTOR and EXP_TYPE in the science data file.

Reference File Format

PHOTOM Reference File Format

Except for MIRI MRS, photom reference files are FITS format with a single BINTABLE extension. The primary data array is always empty. The columns of the table vary with instrument according to the selection criteria listed above. The first few columns always correspond to the selection criteria, such as Filter and Pupil, or Filter and Grating. The remaining columns contain the data relevant to the photometric conversion and consist of PHOTMJSR, UNCERTAINTY, NELEM, WAVELENGTH, and RELRESPONSE. The table column names and data types are listed below.

- FILTER (string) - MIRI, NIRCam, NIRISS, NIRSpec
- PUPIL (string) - NIRCam, NIRISS
- ORDER (integer) - NIRISS
- GRATING (string) - NIRSpec
- SLIT (string) - NIRSpec Fixed-Slit
- SUBARRAY (string) - MIRI Imager/LRS
- PHOTMJSR (float) - all instruments
- UNCERTAINTY (float) - all instruments

- NELEM (int) - if NELEM > 0, then NELEM entries are read from each of the WAVELENGTH and RELRESPONSE arrays
- WAVELENGTH (float 1-D array)
- RELRESPONSE (float 1-D array)

The primary header of the photom reference file 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

The MIRI MRS photom reference files do not contain tabular information, but instead contain the following FITS extensions:

- SCI IMAGE 2D float
- ERR IMAGE 2D float
- DQ IMAGE 2D unsigned-integer
- DQ_DEF TABLE
- PIXSIZ IMAGE 2D float

The SCI extension contains a 2D array of spectral sensitivity factors corresponding to each pixel in a 2D MRS slice image. The sensitivity factors are in units of DN/sec/mJy/pixel. 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, in units of square-arcsec.

The SCI and PIXSIZ array values are both divided into the science product SCI and ERR arrays, yielding image pixels that are 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.

AREA Reference File Format

Pixel area map reference files are FITS format with a single image extension with 'EXTNAME=SCI', which contains a 2-D floating-point array of values. The FITS primary data array is always empty. The primary header contains the keywords PIXAR_SR and PIXAR_A2, which should have the same values as the keywords in the header of the corresponding photom reference file.

Constructing a PHOTOM Reference File

The most straight-forward way to construct a PHOTOM reference file is to populate a photom data model within python and then save the data model to a FITS file. Each instrument has its own photom data model, which contains the columns of information unique to that instrument:

- FgsPhotomModel
- NircamPhotomModel
- NirissPhotomModel
- NirspecPhotomModel (NIRSpec imaging, IFU, MOS)
- NirspecFSPhotomModel (NIRSpec fixed slits)

- MiriImgPhotomModel (MIRI imaging)
- MiriMrsPhotomModel (MIRI MRS)

A NIRISS photom reference file, for example, could be constructed as follows from within the python environment:

```
>>> from jwst import models
>>> import numpy as np
>>> output=models.NirissPhotomModel()
>>> filter=np.array(['F277W','F356W','CLEAR'])
>>> pupil=np.array(['CLEARP','CLEARP','F090W'])
>>> photf=np.array([1.e-15,2.e-15,3.e-15])
>>> uncer=np.array([1.e-17,2.e-17,3.e-17])
>>> nelem=np.zeros(3)
>>> wave=np.zeros(3)
>>> resp=np.zeros(3)
>>> data=np.array(list(zip(filter,pupil,photf,uncer,nelem,wave,resp)),dtype=output.
↪phot_table.dtype)
>>> output.phot_table=data
>>> output.save('niriss_photom_0001.fits')
```

jwst.photom Package

Classes

PhotomStep([name, parent, config_file, ...])	PhotomStep: Module for loading photometric conversion infomation from

PhotomStep

class jwst.photom.**PhotomStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

PhotomStep: Module for loading photometric conversion infomation 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.
- ****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

reference_file_types	

Methods Summary

process(input)	This is where real work happens.

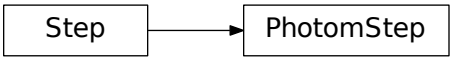
Attributes Documentation

reference_file_types = ['photom', 'area']

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



12.1.37 Pipeline Classes and Configuration Files

Pipeline Modules

The pipelines that call individual correction steps in various orders are defined as python classes within python code modules. The pipelines can be executed by referencing their class name or through the use of a configuration (.cfg) file that in turn references the class. The table below shows the pipeline classes that are currently available, the corresponding pre-defined configurations that make use of those classes, and the instrument modes to which they can be applied.

Class Name	Configuration File	Used For
Detector1Pipeline	calwebb_detector1.cfg	Stage 1: all non-TSO modes
Detector1Pipeline	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
Spec2Pipeline	calwebb_spec2.cfg	Stage 2: spectroscopy modes
Image3Pipeline	calwebb_image3.cfg	Stage 3: imaging modes
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: Time Series mode

The data from different observing modes needs to be processed with different combinations of the pipeline stages listed above. Observing modes are usually identifiable via the value of the `EXP_TYPE` keyword in the data product. The following table lists the pipeline modules that get applied to each `EXP_TYPE` instance.

EXP_TYPE	Stage 1 Pipeline	Stage 2 Pipeline	Stage 3 Pipeline
FGS_IMAGE	calwebb_detector1	calwebb_image2	calwebb_image3
FGS_FOCUS	calwebb_detector1	calwebb_image2	N/A
FGS_DARK	calwebb_dark1	N/A	N/A
FGS_SKYFLAT FGS_INTFLAT	calwebb_detector1	N/A	N/A
MIR_IMAGE	calwebb_detector1	calwebb_image2	calwebb_image3
MIR_MRS	calwebb_detector1	calwebb_spec2	calwebb_spec3
MIR_LRS-FIXEDSLIT	calwebb_detector1	calwebb_spec2	calwebb_spec3
MIR_LRS-SLITLESS	calwebb_tso1	calwebb_spec2	calwebb_tso3
MIR_LYOT MIR_4QPM	calwebb_detector1	calwebb_image2	calwebb_coron3

Continued on next page

Table 272 – continued from previous page

EXP_TYPE	Stage 1 Pipeline	Stage 2 Pipeline	Stage 3 Pipeline
MIR_TACQ	calwebb_detector1	calwebb_image2	N/A
MIR_DARK	calwebb_dark1	N/A	N/A
MIR_FLATIMAGE MIR_FLATMRS	calwebb_detector1	N/A	N/A
NRC_IMAGE	calwebb_detector1	calwebb_image2	calwebb_image3
NRC_CORON	calwebb_detector1	calwebb_image2	calwebb_coron3
NRC_WFSS	calwebb_detector1	calwebb_spec2	calwebb_spec3
NRC_TSIMAGE	calwebb_tso1	calwebb_image2	calwebb_tso3
NRC_TSGRISM	calwebb_tso1	calwebb_spec2	calwebb_tso3
NRC_TACQ NRC_TACONFIRM NRC_FOCUS	calwebb_detector1	calwebb_image2	N/A
NRC_DARK	calwebb_dark1	N/A	N/A
NRC_FLAT NRC_LED	calwebb_detector1	N/A	N/A
NIS_IMAGE	calwebb_detector1	calwebb_image2	calwebb_image3
NIS_WFSS	calwebb_detector1	calwebb_spec2	calwebb_spec3
NIS_SOSS	calwebb_tso1	calwebb_spec2	calwebb_tso3

Continued on next page

Table 272 – continued from previous page

EXP_TYPE	Stage 1 Pipeline	Stage 2 Pipeline	Stage 3 Pipeline
NIS_AMI	calwebb_detector1	calwebb_image2	calwebb_ami3
NIS_TACQ NIS_TACONFIRM NIS_FOCUS	calwebb_detector1	calwebb_image2	N/A
NIS_DARK	calwebb_dark1	N/A	N/A
NIS_LAMP	calwebb_detector1	N/A	N/A
NRS_FIXEDSLIT NRS_IFU NRS_MSASPEC	calwebb_detector1	calwebb_spec2	calwebb_spec3
NRS_BRIGHTOBJ	calwebb_tso1	calwebb_spec2	calwebb_tso3
NRS_IMAGE NRS_TACQ NRS_TACONFIRM NRS_BOTA NRS_TASLIT NRS_CONFIRM NRS_FOCUS NRS_MIMF	calwebb_detector1	calwebb_image2	N/A
NRS_DARK	calwebb_dark1	N/A	N/A
NRS_AUTOWAVE NRS_AUTOFLAT NRS_LAMP	calwebb_detector1	N/A	N/A

Input Files, Output Files and Data Models

An important concept used throughout the JWST pipeline is the *Data Model*. Nearly all data used by any of the pipeline code is encapsulated in a data model. Most input is read into a data model and all output is produced by a data model. When possible, this document will indicate the data model associated with a file type, usually as a parenthetical

link to the data model in question. For some steps, the output file may represent different data models depending on the input to those steps. As a result, the data models listed here will not be an exhaustive list.

Stage 1 Pipeline Step Flow (calwebb_detector1)

Stage 1 processing applies basic detector-level corrections to all exposure types (imaging, spectroscopic, coronagraphic, etc.). It is applied to one exposure at a time. The pipeline module for stage 1 processing is `calwebb_detector1` (the equivalent pipeline class is `Detector1Pipeline`). It is often 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. The list of steps applied by the Build 7.1 calwebb_detector1 pipeline is as follows.

calwebb_detector1 (All Near-IR)	calwebb_detector1 (MIRI)
group_scale	group_scale
dq_init	dq_init
saturation	saturation
ipc	ipc
superbias	linearity
refpix	rscd
linearity	lastframe
persistence	dark_current
dark_current	refpix
	persistence
jump	jump
ramp_fit	ramp_fit
gain_scale	gain_scale

If the `calwebb_tso1.cfg` configuration file is used to execute this pipeline, the `ipc`, `lastframe`, and `persistence` steps will be skipped.

Inputs

- Raw 4D product: The input to `calwebb_detector1` is a single raw exposure file, 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).

Outputs

- 2D Countrate product: All types of inputs result in a 2D countrate product, resulting from 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`.
- 3D Countrate product: If the input exposure contains more than one integration (NINTS>1), 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`.

Arguments

The `calwebb_detector1` pipeline has one optional argument:

- `save_calibrated_ramp`

which is a boolean argument with a default value of `False`. If the user sets it to `True`, the pipeline will save intermediate data to a file as it exists at the end of the `jump` step (just before ramp fitting). 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 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`).

Dark Pipeline Step Flow (calwebb_dark)

The stage 1 dark (`calwebb_dark`) processing pipeline is intended for use with dark exposures. It applies all of the same detector-level correction steps as the `calwebb_detector1` pipeline, but stops just before the application of the `dark_current` step.

Inputs

- Raw 4D Dark product: The input to `calwebb_dark` is a single raw dark exposure.

Outputs

- 4D Corrected product: The output is a 4D (ncols x nrows x ngroups x nints) product that has had all corrections up to, but not including, the `dark_current` step, with a product file type of `_dark`.

Arguments

The `calwebb_dark` pipeline does not have any optional arguments.

Guider Pipeline Step Flow (calwebb_guider)

The guider (`calwebb_guider`) processing pipeline is only for use with FGS guiding mode exposures (ID, ACQ1, ACQ2, TRACK, and FineGuide). It applies three detector-level correction and calibration steps to uncalibrated guider data files, as listed in the table below.

calwebb_guider
dq_init
guider_cds
flat_field

Inputs

- Raw 4D guide-mode product: The input to `calwebb_guider` is a single raw guide-mode data file.

Outputs

- 3D Calibrated product: The output is a 3D (ncols x nrows x nints) countrate product that has been flat-fielded and has bad pixels flagged. See the documentation for the guider_cds step for details on the conversion from raw readouts to countrate images.

Arguments

The `calwebb_guider` pipeline does not have any optional arguments.

Stage 2 Imaging Pipeline Step Flow (calwebb_image2)

Stage 2 imaging (`calwebb_image2`) processing applies additonal corrections that result in a fully calibrated individual exposure. The list of correction steps applied by the calwebb_image2 imaging pipeline is as follows.

calwebb_image2
background
assign_wcs
flat_field
photom
resample

Inputs

- 2D or 3D Countrate product: The input to the `calwebb_image2` pipeline is a countrate exposure, in the form of either `_rate` or `_rateints` files. 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, the steps in the pipeline are applied individually to each integration in an exposure, where appropriate.

Outputs

- 2D or 3D Calibrated product: The output is a calibrated exposure, using the product type suffix `_cal` or `_calints`, depending on the type of input (e.g. `jw80600012001_02101_00003_mirimage_cal.fits`).

Arguments

The `calwebb_image2` pipeline has one optional argument `save_bsub`, which is set to `False` by default. 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 the type of input).

Stage 2 Spectroscopic Pipeline Step Flow (calwebb_spec2)

Stage 2 spectroscopic (`calwebb_spec2`) pipeline applies additional corrections to countrate products that result in fully calibrated individual exposures. The list of correction steps is shown below. Some steps are only applied to certain instruments or instrument modes, as noted in the table.

Instrument Mode	NIRSpec			MIRI			NIRISS		NIRCam
Step	FS	MOS	IFU	FS	SL	MRS	SOSS	WFSS	WFSS
assign_wcs	X	X	X	X	X	X	X	X	X
background	X	X	X	X	X	X	X	X	X
imprint		X	X						
msaflagopen		X	X						
extract_2d[1]	X	X						X	X
flat_field[1]	X	X	X	X	X	X	X	X	X
srctype	X	X	X	X	X	X	X	X	X
straylight						X			
fringe						X			
pathloss	X	X	X						
barshadow		X							
photom	X	X	X	X	X	X	X	X	X
resample_spec	X	X							
cube_build			X			X			
extract_1d	X	X	X	X	X	X	X	X	X

[1]Note that the order of the extract_2d and flat_field steps is reversed (flat_field is performed first) for NIRISS and NIRCam WFSS exposures.

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.

Inputs

The input to the `calwebb_spec2` pipeline can be either a single countrate (`_rate` or `_rateints`) exposure or an Association (ASN) file listing multiple exposures. The background subtraction (`bkg_subtract`) and imprint subtraction (`imprint_subtract`) steps can only be executed when the pipeline is supplied with an association of exposures, because they rely on multiple exposures to perform their tasks. The ASN file must not only list the input exposures, but must also contain information that indicates their relationships to one another.

The background subtraction step can be applied to an assocation containing nodded exposures, such as for MIRI LRS fixed-slit, NIRSpec fixed-slit, and NIRSpec MSA observations, or an association that contains dedicated exposures of a background target. The step will accomplish background subtraction by doing direct subtraction of nodded exposures from one another or by direct subtraction of dedicated background expsoures from the science target exposures.

Background subtraction for Wide-Field Slitless Spectroscopy (WFSS) exposures is accomplished by scaling and subtracting a master background image from a CRDS reference file.

The imprint subtraction step, which is only applied to NIRSpec MSA and IFU exposures, also requires the use of an ASN file, in order to specify which of the inputs is to be used as the imprint exposure. The imprint exposure will be subtracted from all other exposures in the association.

If a single countrate product is used as input, the background subtraction and imprint subtraction steps will be skipped and only the remaining regular calibration steps will be applied to the input exposure.

Outputs

Two or three different types of outputs are created by `calwebb_spec2`.

- Calibrated product: All types of inputs result in a fully-calibrated product at the end of the photom step, which uses the _cal or _calints product type suffix, depending on whether the input was a _rate or _rateints product, respectively.
- Resampled 2D product: If the input is a 2D exposure type that gets resampled/rectified by the resample_spec step, the rectified 2D spectral product created by the resample_spec step is saved as a _s2d file. 3D (_rateints) input exposures do not get resampled.
- Resampled 3D product: If the data are NIRSpec IFU or MIRI MRS, the result of the cube_build step will be saved as a _s3d file.
- 1D Extracted Spectrum product: 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.

If the input to calwebb_spec2 is an ASN file, these products are created for each input exposure.

Arguments

The calwebb_spec2 pipeline has one optional argument:

- save_bsub

which is a Boolean argument with a default value of False. If the user sets it to True, the results of the background subtraction step (if applied) are saved to an intermediate file of type _bsub or _bsubints, as appropriate.

Stage 3 Imaging Pipeline Step Flow (calwebb_image3)

Stage 3 processing for 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.

calwebb_image3
tweakreg
skymatch
outlier_detection
resample
source_catalog

Inputs

- Associated 2D Calibrated products: The inputs to calwebb_image3 will usually be in the form of an ASN file that lists multiple exposures to be processed and combined into a single output product. The individual exposures should be calibrated (_cal) products from calwebb_image2 processing.
- Single 2D Calibrated product: 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

- Resampled 2D Image product (*DrizProductModel*): A resampled/rectified 2D image product of type _i2d is created containing the rectified single exposure or the rectified and combined association of exposures, which is the direct output of the resample step.

- Source catalog: A source catalog produced from the _i2d product is saved as an ASCII file in ecsv format, with a product type of _cat.
- CR-flagged products: If the outlier_detection step is applied, a new version of each input calibrated exposure 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 the file is identified by including the association candidate ID in the original input _cal file name and changing the product type to _crf, e.g. jw96090001001_03101_00001_nrca2_o001_crf.fits.

Stage 3 Spectroscopic Pipeline Step Flow (calwebb_spec3)

Stage 3 processing for spectroscopic observations is intended for combining the calibrated data from multiple exposures (e.g. a dither pattern) into a single rectified (distortion corrected) product and a combined 1D spectrum. Before being combined, the exposures may receive additional corrections for the purpose of background matching and outlier rejection. The steps applied by the calwebb_spec3 pipeline are shown below.

Instrument Mode	NIRSpec			MIRI		NIRISS	NIRCam
Step	FS	MOS	IFU	FS	MRS	WFSS	WFSS
mrs_imatch					X		
outlier_detection	X	X	X	X	X	X	X
resample_spec	X	X		X		X	X
cube_build			X		X		
extract_1d	X	X	X	X	X	X	X

NOTE: In B7.1 the calwebb_spec3 pipeline is very much a prototype and not all steps are functioning properly for all modes. In particular, the outlier_detection step does not yet work well, if at all, for any of the spectroscopic modes. Also, the resample_spec step does not work for dithered slit-like spectra (i.e. all non-IFU modes). Processing of NIRSpec IFU and MIRI MRS exposures does work, using the mrs_imatch, cube_build, and extract_1d steps.

Inputs

- Associated 2D Calibrated products: The inputs to calwebb_spec3 will usually be in the form of an ASN file that lists multiple exposures to be processed and combined into a single output product. The individual exposures should be calibrated (_cal) products from calwebb_spec2 processing.

Outputs

- CR-flagged products: If the outlier_detection step is applied, a new version of each input calibrated exposure 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 the file is identified by including the association candidate ID in the original input _cal file name and changing the product type to _crf, e.g. jw96090001001_03101_00001_nrs2_o001_crf.fits.
- Resampled 2D spectral product (*DrizProductModel*): A resampled/rectified 2D product of type _s2d is created containing the rectified and combined association of exposures, which is the direct output of the resample_spec step, when processing non-IFU modes.
- Resampled 3D spectral product (*IFUCubeModel*): A resampled/rectified 3D product of type _s3d is created containing the rectified and combined association of exposures, which is the direct output of the cube_build step, when processing IFU modes.

- 1D Extracted Spectrum product: All types of inputs result in a 1D extracted spectral data product, which is saved as a _x1d file, and is the result of performing 1D extraction on the combined _s2d or _s3d product.

Stage 3 Aperture Masking Interferometry (AMI) Pipeline Step Flow (calwebb_ami3)

The stage 3 AMI (calwebb_ami3) pipeline is to be applied to associations of calibrated NIRISS AMI exposures and is used to compute fringe parameters and correct science target fringe parameters using observations of reference targets. The steps applied by the calwebb_ami3 pipeline are shown below.

calwebb_ami3
ami_analyze
ami_average
ami_normalize

Inputs

- Associated 2D Calibrated products: The inputs to calwebb_ami3 need to be in the form of an ASN file that lists multiple science target exposures, and optionally reference target exposures as well. The individual exposures should be in the form of calibrated (_cal) products from calwebb_image2 processing.

Outputs

- AMI product (*AmiLgModel*): For every input exposure, the fringe parameters and closure phases caculated by the ami_analyze step are saved to an _ami product file, which is a table containing the fringe parameters and closure phases. Product names use the original input _cal 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 AMI product (*AmiLgModel*): The AMI results averaged over all science or reference exposures, calculated by the ami_average step, are saved to an _amiavg product file. Separate products are created for the science target and reference target data. Note that these output products are only created if the pipeline argument save_averages (see below) is set to True.
- Normalized AMI product (*AmiLgModel*): If reference target exposures are included in the input ASN, the averaged AMI results for the science target will be normalized by the averaged AMI results for the reference target, via the ami_normalize step, and will be saved to an _aminorm product file.

Arguments

The calwebb_ami3 pipeline has one optional argument:

- save_averages

which is a Boolean parameter set to a default value of False. If the user sets this agument to True, the results of the ami_average step will be saved, as described above.

Stage 3 Coronagraphic Pipeline Step Flow (calwebb_coron3)

The stage 3 coronagraphic (calwebb_coron3) 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
stack_refs
align_refs
klip
outlier_detection
resample

Inputs

- Associated Calibrated products: The input to calwebb_coron3 is assumed to be in the form of an ASN file that lists multiple observations of a science target and, optionally, a reference PSF target. The individual science target and PSF reference exposures should be in the form of 3D calibrated (_calints) products from calwebb_image2 processing.

Outputs

- Stacked PSF images: The data from each input PSF reference exposure are concatenated into a single combined 3D stack, for use by subsequent steps. The stacked PSF data gets written to disk in the form of a psfstack (_psfstack) product from *stack_refs step*.
- Aligned PSF images: The initial processing requires aligning all input PSFs specified in the ASN. The aligned PSF images then gets written to disk in the form of psfalign (_psfalign) products from *align_refs step*.
- PSF-subtracted exposures: The *klip step* takes the aligned PSF images and applies them to each of the science exposures in the ASN to create psfsub (_psfsub) products.
- CR-flagged products: The *OutlierDetectionStep* step is applied to the psfsub products to flag pixels in the DQ array that have been detected as outliers. This updated product is known as a CR-flagged product. A outlier-flagged product of type _crfints is created and can optionally get written to disk.
- Resampled product: The *resample step* is applied to the CR-flagged products to create a single resampled, combined product for the science target. This resampled product of type _i2d gets written to disk and returned as the final product from this pipeline.

Stage 3 Time-Series Observation(TSO) Pipeline Step Flow (calwebb_tso3)

The stage 3 TSO (calwebb_tso3) pipeline is to be applied to associations of calibrated TSO exposures (NIRCam TS imaging, NIRCam TS grism, NIRISS SOSS, NIRSpec BrightObj, MIRI LRS Slitless) and is used to produce calibrated time-series photometry of the source object.

The steps applied by the calwebb_tso3 pipeline for an Imaging TSO observation are shown below:

calwebb_tso3
outlier_detection
tso_photometry

The steps applied by the calwebb_tso3 pipeline for a Spectroscopic TSO observation are shown below:

calwebb_tso3
outlier_detection
extract_1d
white_light

Inputs

- Associated 3D Calibrated products: The input to `calwebb_tso3` is assumed to be in the form of an ASN file that lists multiple science observations of a science target. The individual exposures should be in the form of 3D calibrated (`_calints`) products from either `calwebb_image2` or `calwebb_spec2` processing.

Outputs

- CR-flagged products: If the `OutlierDetectionStep` step is applied, a new version of each input calibrated exposure product is created, which contains a DQ array that has been updated to flag pixels detected as outliers. This update product is known as a CR-flagged product. A outlier-flagged product of type `_crfints` is created and can optionally get written to disk.
- Source photometry catalog for imaging TS observations: A source catalog produced from the `_crfints` product is saved as an ASCII file in `ecsv` format with a product type of `_phot`.
- Extracted 1D spectra for spectroscopic TS observations: The `extract_1d` step is applied to create a `MultiSpecModel` for the entire set of spectra, with a product type of `_x1dints`.
- White-light photometry for spectroscopic TS observations: The `white_light` step is applied to the `_x1dints` extracted data to produce an ASCII catalog in `ecsv` format with a product type of `_whtlt`, containing the wavelength-integrated white-light photometry of the source object.

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.

Continued on next page

Table 273 – continued from previous page

Spec3Pipeline(*args, **kwargs)	Spec3Pipeline: Processes JWST spectroscopic exposures from Level 2b to 3.
TestLinearPipeline(*args, **kwargs)	See `Step.__init__` for the parameters.
Tso3Pipeline(*args, **kwargs)	TSO3Pipeline: Applies level 3 processing to TSO-mode data from

Ami3Pipeline

class jwst.pipeline.**Ami3Pipeline**(**args*, ***kwargs*)

Bases: *jwst.stpipe.Pipeline*

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

spec	
step_defs	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

spec = '\n save_averages = boolean(default=False)\n '

step_defs = {'ami_analyze': <class 'jwst.ami.ami_analyze_step.AmiAnalyzeStep'>, 'ami_

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: *jwst.stpipe.Pipeline*

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

spec
step_defs

Methods Summary

process(input)	Primary method for performing pipeline.

Attributes Documentation

spec = "\n suffix = string(default='i2d')\n "

step_defs = {'align_refs': <class 'jwst.coron.align_refs_step.AlignRefsStep'>, 'klip'

Methods Documentation

process(*input*)
Primary method for performing pipeline.

DarkPipeline

class jwst.pipeline.**DarkPipeline**(**args*, ***kwargs*)
Bases: *jwst.stpipe.Pipeline*

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

step_defs

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

step_defs = {'dq_init': <class 'jwst.dq_init.dq_init_step.DQInitStep'>, '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.

Detector1Pipeline

class jwst.pipeline.**Detector1Pipeline**(**args*, ***kwargs*)
Bases: *jwst.stpipe.Pipeline*

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

spec	
step_defs	

Methods Summary

process(input)	This is where real work happens.
setup_output(input)	

Attributes Documentation

spec = '\n save_calibrated_ramp = boolean(default=False)\n '

step_defs = {'dark_current': <class 'jwst.dark_current.dark_current_step.DarkCurrentS

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: *jwst.stpipe.Pipeline*

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

step_defs	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

step_defs = {'dq_init': <class 'jwst.dq_init.dq_init_step.DQInitStep'>, 'flat_field':

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: *jwst.stpipe.Pipeline*

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

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

image_exptypes = ['MIR_IMAGE', 'NRC_IMAGE', 'NIS_IMAGE']

spec = '\n save_bsub = boolean(default=False) # Save background-subracted science\n '

step_defs = {'assign_wcs': <class 'jwst.assign_wcs.assign_wcs_step.AssignWcsStep'>, '

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: *jwst.stpipe.Pipeline*

Image3Pipeline: Applies level 3 processing to imaging-mode data from any JWST instrument.

Included steps are: tweakreg skymatch outlier_detection resample source_catalog

See `Step.__init__` for the parameters.

Attributes Summary

spec
step_defs

Methods Summary

process(input)	Run the Image3Pipeline

Attributes Documentation

spec = "\n suffix = string(default='i2d')\n "

step_defs = {'outlier_detection': <class 'jwst.outlier_detection.outlier_detection_st

Methods Documentation

process(*input*)
Run the Image3Pipeline

Parameters **input** (*Level3 Association, or ModelContainer*) – The exposures to process

Spec2Pipeline

class jwst.pipeline.**Spec2Pipeline**(**args*, ***kwargs*)

Bases: *jwst.stpipe.Pipeline*

Spec2Pipeline: Processes JWST spectroscopic exposures from Level 2a to 2b. Accepts a single exposure or an association as input.

Included steps are: assign_wcs, background subtraction, NIRSpec MSA imprint subtraction, NIRSpec MSA bad shutter flagging, 2-D subwindow extraction, flat field, source type decision, straylight, fringe, pathloss, barshadow, photom, resample_spec, cube_build, and extract_1d.

See `Step.__init__` for the parameters.

Attributes Summary

spec
step_defs

Methods Summary

process(input)	Entrypoint for this pipeline
process_exposure_product(exp_product[, ...])	Process an exposure found in the association product

Attributes Documentation

spec = '\n save_bsub = boolean(default=False) # Save background-subracted science\n fa

step_defs = {'assign_wcs': <class 'jwst.assign_wcs.assign_wcs_step.AssignWcsStep'>, '

Methods Documentation

process(*input*)

Entrypoint for this pipeline

Parameters **input** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *Level2 Association*, *or* DataModel) – 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: *jwst.stpipe.Pipeline*

Spec3Pipeline: Processes JWST spectroscopic exposures from Level 2b to 3.

Included steps are: MIRI MRS background matching (skymatch) outlier detection (outlier_detection) 2-D spectroscopic resampling (resample_spec) 3-D spectroscopic resampling (cube_build) 1-D spectral extraction (extract_1d)

See `Step.__init__` for the parameters.

Attributes Summary

spec
step_defs

Methods Summary

process(input)	Entrypoint for this pipeline

Attributes Documentation

spec = '\n '

step_defs = {'cube_build': <class 'jwst.cube_build.cube_build_step.CubeBuildStep'>, '

Methods Documentation

process(*input*)
Entrypoint for this pipeline

Parameters **input** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *Level3 Association*, *or* DataModel) – The exposure or association of exposures to process

TestLinearPipeline

class jwst.pipeline.**TestLinearPipeline**(**args*, ***kwargs*)
Bases: *jwst.stpipe.LinearPipeline*

See `Step.__init__` for the parameters.

Attributes Summary

pipeline_steps
step_defs

Attributes Documentation

pipeline_steps = [('ipc', <class 'jwst.ipc.ipc_step.IPCStep'>), ('dq_init', <class 'jw

step_defs = {'assign_wcs': <class 'jwst.assign_wcs.assign_wcs_step.AssignWcsStep'>, '

Tso3Pipeline

class jwst.pipeline.**Tso3Pipeline**(**args*, ***kwargs*)

Bases: *jwst.stpipe.Pipeline*

TSO3Pipeline: Applies level 3 processing to TSO-mode data from any JWST instrument.

Included steps are:

- outlier_detection
- tso_photometry
- extract_1d
- white_light

See `Step.__init__` for the parameters.

Attributes Summary

image_exptypes	
reference_file_types	
spec	
step_defs	

Methods Summary

process(input)	Run the TSO3Pipeline

Attributes Documentation

image_exptypes = ['NRC_TSIMAGE']

reference_file_types = ['gain', 'readnoise']

spec = '\n scale_detection = boolean(default=False)\n '

step_defs = {'extract_1d': <class 'jwst.extract_1d.extract_1d_step.Extract1dStep'>, '

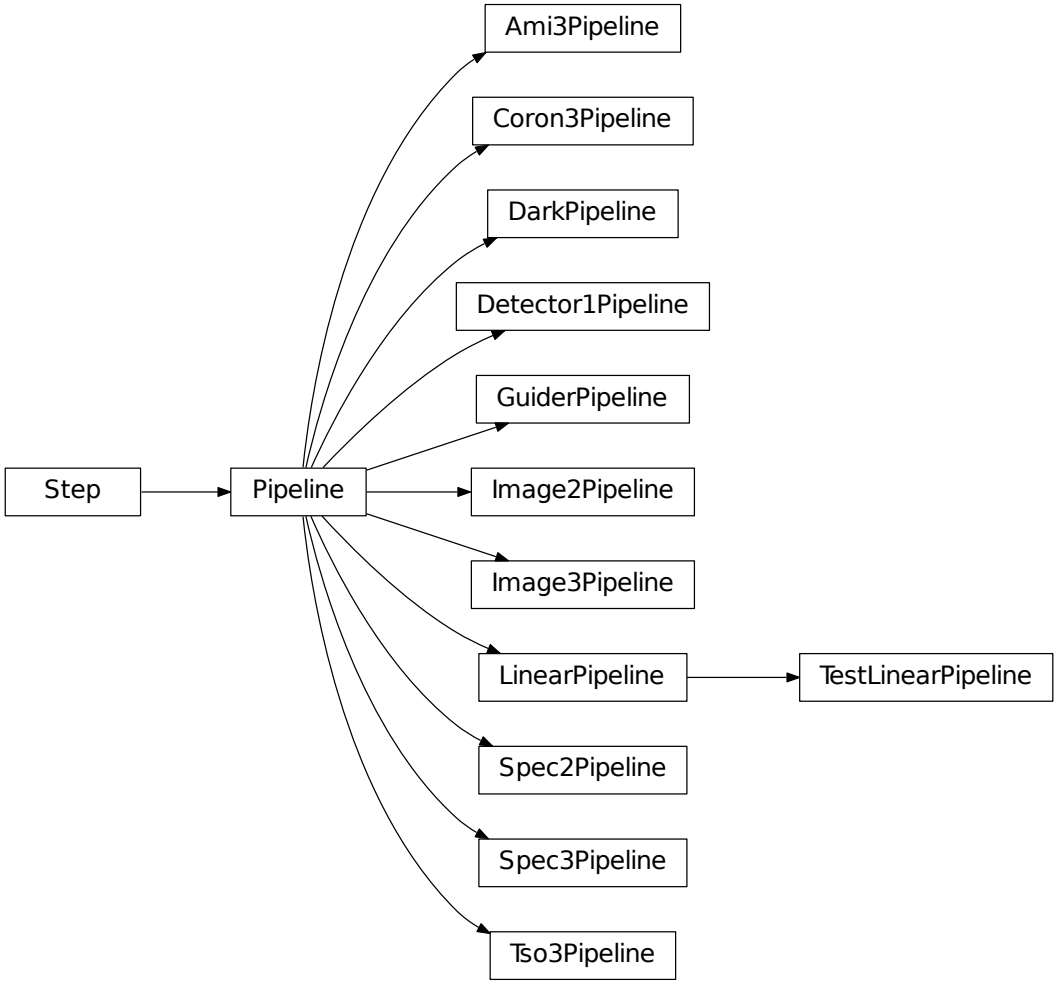
Methods Documentation

process(*input*)

Run the TSO3Pipeline

Parameters **input** (*Level3 Association, json format*) – The exposures to process

Class Inheritance Diagram



12.1.38 Ramp Fitting

Description

This step determines the mean count rate for each pixel by performing a linear fit to the data in the input (jump) file. The fit is done using "ordinary least squares" (the "generalized least squares" is no longer an option). The fit is performed independently for each pixel. There are up to three output files. The primary output file, giving the slope at each pixel, is always produced. If the input exposure contains more than one integration, the resulting slope images from each integration are stored as a data cube in a second output data product. A third, optional output product is also available and is produced only when the step parameter 'save_opt' is True (the default is False). The output values will be in units of counts per second. Following a description of the fitting algorithm, these three type of output files are detailed 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 this interval will be referred to as a "segment". The fitting algorithm does an 'optimal' linear fit, which is the weighting used by Fixsen et al, PASP,112, 1350. ('unweighted' in which all groups for a pixel are equally weighted, is no longer a weighting option.) Segments are derived using the 4-D GROUPDQ array of the input data set, under the assumption that the jump step will have already flagged CR's. Segments are also 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.

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 to be 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 of the 2 valid values of the science data. If any input dataset contains ramps saturated in their second group, the count rates for those pixels in that integration will be calculated to be the value of the science data in the first group divided by the group time. 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 to a file as the primary output product. In this output product, the 4-D GROUPDQ from all integrations is compressed into 2-D, which is then merged (using a bitwise OR) with the input 2-D PIXELDQ to create the output DQ array. The 3-D VAR_POISSON and VAR_RNOISE arrays from all integrations are averaged into corresponding 2-D output arrays. If the ramp fitting step is run by itself, the output file name will have the suffix '_RampFit' or the suffix '_RampFitStep'; if the ramp fitting step is run as part of the calwebb_detector1 pipeline, the final output file name will have the suffix '_rate'. In either case, the user can override this name by specifying an output file name.

If the input exposure contains more than one integration, the resulting slope images from each integration are stored as a data cube in a second output data product. Each plane of the 3-D SCI, ERR, DQ, VAR_POISSON, and VAR_RNOISE arrays in this product is the result for a given integration. In this output product, the GROUPDQ data for a given integration is compressed 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 from an integration are calculated by averaging over the fit segments in the corresponding 4-D arrays. By default, the name of this output product is based on the name of the input file and will have the suffix '_rateints'; the user can override this name by specifying a name using the parameter int_name.

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 which 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. (Calculaton of the two variance arrays requires retrieving readnoise and gain values from their respective reference files.) The y-intercept refers to the result of the fit at an 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 is based on the name of the input file and will have the suffix '_fitopt'; the user can override this name by specifying a name using the parameter opt_name. 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 of 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.

Slopes and their variances are calculated for each segment, for each integration, and for the entire dataset. As defined above, a segment is a set of contiguous groups where none of the groups are saturated or cosmic ray-affected. 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.

Segment-specific computations:

The slope of each segment is calculated using the least-squares method with optimal weighting. The variance of the slope of the 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 variance of the slope of the 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 the 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 the 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 the 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 the integration is due to both Poisson and read noise:

$$var_i^C = \frac{1}{\sum_s \frac{1}{var_s^R + var_s^P}}$$

The slope for the 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 what gets 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:

$$slope_o = \frac{\sum_{i,s} \frac{slope_{i,s}}{var_{i,s}^C}}{\sum_{i,s} \frac{1}{var_{i,s}^C}}$$

Upon successful completion of this step, the status keyword S_RAMP will be set to COMPLETE.

The MIRI first frame correction step flags all pixels in the first group of data in each integration of a MIRI exposure having more than 3 groups, 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 data in each integration of a MIRI exposure having more than 2 groups, so that those data do not get used in either the jump detection or ramp fitting steps. The ramp fitting will only fit data if there are at least 2 good groups of data, and will log a warning otherwise.

Step 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 optional output information.
- `--opt_name`: A string that can be used to override the default name for the optional output information.
- `--int_name`: A string that can be used to override the default name for the integration-by-integration slopes, for the case that the input file contains more than one integration.

Reference Files

The Ramp Fitting step uses two reference files: GAIN and READNOISE. 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 read noise values are used as part of the noise estimate for each pixel. Both are necessary for proper computation of noise estimates.

GAIN Reference Files

The GAIN reference file is selected based on instrument, detector and, where necessary, subarray.

READNOISE Reference Files

The READNOISE reference file is selected by instrument, detector and, where necessary, subarray.

Reference File Formats

GAIN Reference Files

The gain reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of gain values (in e/DN) per pixel. The REFTYPE value is `GAIN`.

READNOISE Reference Files

The read noise reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of read noise values per pixel. The units of the read noise should be DN and should be the CDS (Correlated Double Sampling) read noise, i.e. the effective noise between any pair of non-destructive detector reads. The REFTYPE value is `READNOISE`.

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: *jwst.stpipe.Step*

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.
- ****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
reference_file_types
spec

Continued on next page

Table 296 – continued from previous page

weighting

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

algorithm = 'ols'

reference_file_types = ['readnoise', 'gain']

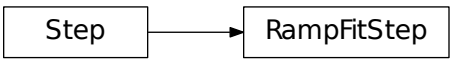
spec = "\n int_name = string(default='')\n save_opt = boolean(default=False) # Save op

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



12.1.39 JWST Calibration Reference File Formats

Introduction

Purpose and Scope

This document specifies the format of each calibration reference file used by the JWST Calibration pipeline for DMS Build 7, satisfying requirement DMS-653 (“The format of each calibration reference file shall be specified in the JWST Calibration Reference File Specification Document.”) The corresponding code internal delivery was packaged as Release Candidate 7 of Build 7. Many calibration steps in the DMS Build 7 Calibration Pipeline require reference files retrieved from CRDS. This document is intended to be a reference guide to the formats of reference files for steps requiring them, and is not intended to be a detailed description of each of those pipeline steps.

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 1, 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 1. Flags for the 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	RESERVED	
5	32	RESERVED	
6	64	RESERVED	
7	128	RESERVED	
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	UNRELIABLE_RESET	Sensitive to reset anomaly
29	536870912	MSA_FAILED_OPEN	Pixel sees light from failed-open shutter
30	1073741824	OTHER_BAD_PIXEL	A catch-all flag

Calibration Steps Using Reference Files

AMI Analyse

This step applies the Lacour-Greenbaum (LG) image plane modeling algorithm to a NIRISS Aperture Masking Interferometry (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.

Reference File Types

The ami_analyze step uses a THROUGHPUT reference file, which contains throughput data for the filter used in the input AMI image. (The ami_average and ami_normalize steps do not use any reference files.)

CRDS Selection Criteria

Throughput reference files are selected on the basis of INSTRUME and FILTER values for the input science data set.

Throughput Reference File Format

Throughput reference files are FITS files with one BINTABLE extension. The FITS primary data array is assumed to be empty. The table extension uses `EXTNAME=THROUGHPUT` and the data table has the following characteristics:

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

Assign_wcs

assign_wcs creates and assigns a WCS object to observations. The WCS object is a pipeline of transforms from detector to world coordinates. It may include intermediate coordinate frames and the corresponding transformations between them. The transforms are stored in reference files in CRDS.

Reference File Types

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.

There are 16 reference types used by assign_wcs:

reftype	description	Instrument
camera	NIRSPEC Camera model	NIRSPEC
collimator	NIRSPEC Collimator Model	NIRSPEC
disperser	Disperser parameters	NIRSPEC
distortion	Spatial distortion model	MIRI, FGS, NIRCAM, NIRISS
filteroffset	MIRI Imager filter offsets	MIRI
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
v2v3	Transform from MIRI instrument focal plane to V2V3 plane	MIRI
wavelength-range	Typical wavelength ranges	MIRI, NIRSPEC

CRDS Selection Criteria For Each Reference File Type

CAMERA

CAMERA reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

COLLIMATOR

For NIRSPEC, COLLIMATOR reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

DISPERSER

For NIRSPEC, DISPERSER reference files are currently selected based on the values of EXP_TYPE and GRATING in the input science data set.

DISTORTION

For MIRI, DISTORTION reference files are currently selected based on the values of EXP_TYPE, DETECTOR, CHANNEL and BAND in the input science data set.

For FGS, DISTORTION reference files are currently selected based on the values of EXP_TYPE and DETECTOR in the input science data set.

For NIRCAM, DISTORTION reference files are currently selected based on the values of EXP_TYPE, DETECTOR, and CHANNEL in the input science data set.

For NIRISS, DISTORTION reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

FILTEROFFSET

For MIRI, FILTEROFFSET reference files are currently selected based on the values of EXP_TYPE and DETECTOR in the input science data set.

FORE

For NIRSPEC, FORE reference files are currently selected based on the values of EXP_TYPE and FILTER in the input science data set.

FPA

For NIRSPEC, FPA reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

IFUFORE

For NIRSPEC, IFUFORE reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

IFUPOST

For NIRSPEC, IFUPOST reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

IFUSLICER

For NIRSPEC, IFUSLICER reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

MSA

For NIRSPEC, MSA reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

OTE

For NIRSPEC, OTE reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

SPECWCS

For MIRI, SPECWCS reference files are currently selected based on the values of DETECTOR, CHANNEL, BAND, SUBARRAY, and EXP_TYPE in the input science data set.

For NIRISS, SPECWCS reference files are currently selected based on the values of SUBARRAY and EXP_TYPE in the input science data set.

REGIONS

For MIRI, REGIONS reference files are currently selected based on the values of DETECTOR, CHANNEL, BAND, EXP_TYPE in the input science data set.

V2V3

For MIRI, V2V3 reference files are currently selected based on the values of DETECTOR, CHANNEL, BAND, EXP_TYPE in the input science data set.

WAVELENGTHRANGE

For NIRSPEC, WAVELENGTHRANGE reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

For MIRI, WAVELENGTHRANGE reference files are currently selected based only on the value of EXP_TYPE in the input science data set.

Reference File Formats For Each Reference File Type

CAMERA

This reference file contains an astropy compound model made up of a polynomial models, rotation and translations. The forward direction is from the FPA to the GWA. :model: Transform through the CAMERA.

COLLIMATOR

The collimator reference file contains an astropy compound model made up of a polynomial models, rotation and translations. The forward direction is from the GWA to the MSA.

model Transform through the COLLIMATOR.

DISPERSER

The disperser file contains reference data about the NIRSPEC dispersers (gratings or the prism). The reference data is described in the NIRSPEC Interface Control Document.

The following fields are common for all gratings and the prism:

grating Name of grating

gwa_tiltx

temperatures Temperatures measured where the GWA sensor is

zeroreadings Value of GWA sensor reading which corresponds to disperser model parameters

tilt_model Model of the relation between THETA_Y vs GWA_X reading

gwa_tilty

temperatures Temperatures measured where the GWA sensor is

zeroreadings Value of GWA sensor reading which corresponds to disperser model parameters

tilt_model Model of the relation between THETA_X vs GWA_Y reading

tilt_x Angle (in degrees) between the grating surface and the reference surface (the mirror)

tilt_y Angle (in degrees) between the grating surface and the reference surface (the mirror)

theta_x Element alignment angle in x-axis (in degrees)

theta_y Element alignment angle in y-axis (in degrees)

theta_z Element alignment angle in z-axis (in degrees)

The prism reference file has in addition the following fields:

angle Angle between the front and back surface of the prism (in degrees)

kcoef K coefficients of Selmeir equation, describing the material

lcoef L coefficients describing the material

tcoef Thermal coefficients describing the properties of the glass

tref Reference temperature (in K)

pref Reference pressure (in ATM)

wbound Min and Max wavelength (in meters) for which the model is valid

DISTORTION

The distortion reference file contains a combination of astropy models. For the MIRI Imager this file contains a polynomial and filter-dependent offsets. For the MIRI MRS, NIRCAM, NIRISS, and FGS the model is a combination of polynomials. :model: Transform from detector to an intermediate frame (instrument dependent).

FILTEROFFSET

The filter offset reference file is an ASDF file that contains a dictionary of row and column offsets for the MIRI imaging dataset. The filter offset reference file contains a dictionary in the tree that is indexed by the instrument filter. Each filter points to two fields - row_offset and column_offset. The format is

miri_filter_name

column_offset Offset in x (in arcmin)

row_offset Offset in y (in arcmin)

FORE

The FORE reference file stores the transform through the Filter Wheel Assembly (FWA). It has two fields - "filter" and "model". The transform through the FWA is chromatic. It is represented as a Polynomial of two variables whose coefficients are wavelength dependent. The compound model takes three inputs - x, y positions and wavelength.

filter Filter name.

model Transform through the Filter Wheel Assembly (FWA).

FPA

The FPA reference file stores information on the metrology of the Focal Plane Array (FPA) which consists of two single chip arrays (SCA), named NRS1 and NRS2.

The reference file contains two fields : "NRS1" and "NRS2". Each of them stores the transform (shift and rotation) to transform positions from the FPA to the respective SCA. The output units are in pixels.

NRS1 Transform for the NRS1 detector.

NRS2 Transform for the NRS2 detector.

IFUFORE

The IFU reference file provides the parameters (Paraxial and distortions coefficients) for the coordinate transforms from the MSA plane to the plane of the IFU slicer. :model: Compound model, Polynomials

IFUPOST

The IFUPOST reference file provides the parameters (Paraxial and distortions coefficients) for the coordinate transforms from the slicer plane to the MSA plane (out), that is the plane of the IFU virtual slits.

The reference file contains models made up based on an offset and a polynomial. There is a model for each of the slits and is indexed by the slit number. The models is used as part of the conversion from the GWA to slit.

ifu_slice_number

model Polynomial and rotation models.

IFUSLICER

The IFUSLICER stores information about the metrology of the IFU slicer - relative positioning and size of the aperture of each individual slicer and the absolute reference with respect to the center of the field of view. The reference file contains two fields - "data" and "model". The "data" field is an array with 30 rows pertaining to the 30 slices and the columns are

data Array with reference data for each slicer. It has 5 columns

NO Slice number (0 - 29)

x_center X coordinate of the center (in meters)

y_center Y coordinate of the center (in meters)

x_size X size of the aperture (in meters)

y_size Y size of the aperture (in meters)

model Transform from relative positions within the IFU slicer to absolute positions within the field of view. It's a combination of shifts and rotation.

MSA

The MSA reference file contains information on the metrology of the microshutter array and the associated fixed slits - relative positioning of each individual shutter (assumed to be rectangular) And the absolute position of each quadrant within the MSA.

The MSA reference file has 5 fields, named

1

data Array with reference data for each shutter in Quadrant 1. It has 5 columns

NO Shutter number (1- 62415)

x_center X coordinate of the center (in meters)

y_center Y coordinate of the center (in meters)

x_size X size of the aperture (in meters)

y_size Y size of the aperture (in meters)

model Transform from relative positions within Quadrant 1 to absolute positions within the MSA

2

data Array with reference data for shutters in Quadrant 2, same as in 1 above

model Transform from relative positions within Quadrant 2 to absolute positions within the MSA

3

data Array with reference data for shutters in Quadrant 3, same as in 1 above

model Transform from relative positions within Quadrant 3 to absolute positions within the MSA

4

data Array with reference data for shutters in Quadrant 4, same as in 1 above

model Transform from relative positions within Quadrant 4 to absolute positions within the MSA

5

data Reference data for the fixed slits and the IFU, same as in 1, except NO is 6 rows (1-6) and the mapping is 1 - S200A1, 2 - S200A1, 3 - S400A1, 4 - S200B1, 5 - S1600A1, 6 - IFU

model Transform from relative positions within each aperture to absolute positions within the MSA

OTE

This reference file contains a combination of astropy models - polynomial, shift, rotation and scaling.

model Transform through the Telescope Optical Element (OTE), from the FWA to XAN, YAN telescope frame. The output units are in arcsec.

SPECWCS

For the MIRI LRS mode the file is in FITS format. The reference file contains the zero point offset for the slit relative to the full field of view. For the Fixed Slit exposure type the zero points in X and Y are stored in the header of the second HDU in the 'IMX' and 'IMY' keywords. For the Slitless exposure type they are stored in the header of the

second HDU in FITS keywords 'IMXSLTl' and 'IMYSLTl'. For both of the exposure types, the zero point offset is 1 based and the X (e.g., IMX) refers to the column and Y refers to the row.

For the MIRI MRS the file is in ASDF format with the following structure.

channel The MIRI channels in the observation, e.g. "12".

band The band for the observation (one of "LONG", "MEDIUM", "SHORT").

model

slice_number The wavelength solution for each slice. <slice_number> is the actual slice number (s), computed by s = channel * 100 + slice

For NIRISS SOSS mode the file is in ASDF format with the following structure.

model A tabular model with the wavelength solution.

REGIONS

The IFU takes a region reference file that defines the region over which the WCS is valid. The reference file should define a polygon and may consist of a set of X,Y coordinates that define the polygon.

channel The MIRI channels in the observation, e.g. "12".

band The band for the observation (one of "LONG", "MEDIUM", "SHORT").

regions An array with the size of the MIRI MRS image where pixel values map to the MRS slice number. 0 indicates a pixel is not within any slice.

V2V3

The model field in the tree contains N models, one per channel, that map the spatial coordinates from alpha, beta to XAN, YAN.

channel The MIRI channels in the observation, e.g. "12".

band The band for the observation (one of "LONG", "MEDIUM", "SHORT").

model

channel_band Transform from alpha, beta to XAN, YAN for this channel.

WAVELENGTHRANGE

For MIRI MRS the wavelengthrange file consists of two fields which define the wavelength range for each combination of a 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.

For NIRSPEC the 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

Cal_Ver

The Cal_Ver mechanism is used to track software versions of each of the calibration steps run in the pipeline, primarily for archiving purposes.

Reference File Types

Cal_Ver uses a CALVER reference file.

CRDS Selection Criteria

The CALVER reference files are selected by matching a dataset header against a tuple which defines multiple parameter values whose names are specified in the rmap header parkey.

CALVER Reference File Format

CALVER reference files are json files, containing a version number for each calibration step. The files apply to any steps that are in any imaging or spectroscopic pipeline.

Dark current

The dark current step removes dark current from a JWST exposure by subtracting dark current data stored in a dark reference file. The reference file records a high signal-to-noise ramp of the detector dark signal (i.e., the signal detected in the absence of photons from the sky). It is constructed by averaging the individual frames of many long, dark exposures.

Reference File Types

The dark current step uses a DARK reference file.

CRDS Selection Criteria

Dark reference files are selected on the basis of INSTRUME, DETECTOR, and SUBARRAY values for the input science data set. For MIRI exposures, the value of READPATT is used as an additional selection criterion.

DARK Reference File Format

Dark 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 extensions for the NIR detectors are as follows:

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

The dark reference files for the MIRI detectors depend on the integration number. The first integration dark contains effects from the reset and are slightly different from the other integrations. Currently the MIRI dark reference files only contain the correction for two integrations. The second integration dark can be subtracted from all integrations after the first one. The format of the MIRI dark reference files are as follows:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x ngroups x nints	float
ERR	3	ncols x nrows x ngroups x nints	float
DQ	2	ncols x nrows x 1 x nints	integer

The BINTABLE extension contains the bit assignments used in the DQ array. It uses `EXTNAME=DQ_DEF` and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Data Quality Initialization

The DQ initialization step propagates pixel-dependent flags from a static pixel mask reference file into the 2-D DQ array of the science data. The 2-D pixel mask is first translated from the 8-bit DQ array of the reference file into the 32-bit DQ array specified by the master DQ list, then propagated into the 2-D DQ array of the science data file using a bit-wise OR operation.

Reference File Types

The Data Quality Initialization step uses a MASK reference file.

CRDS Selection Criteria

MASK reference files are currently selected based only on the value of DETECTOR in the input science data set. There is one MASK reference file for each JWST instrument detector.

MASK Reference File Format

The MASK reference file is a FITS file with a primary HDU, 1 IMAGE extension HDU and 1 BINTABLE extension. The primary data array is assumed to be empty. The MASK data are stored in the first IMAGE extension, which shall have EXTNAME='DQ'. The data array in this extension has integer data type and is 2-D, with dimensions equal to the number of columns and rows in a full frame raw readout for the given detector, including reference pixels. Note that this does not include the reference output for MIRI detectors.

The BINTABLE extension contains the bit assignments used in the DQ array. It uses `EXTNAME=DQ_DEF` and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition

- DESCRIPTION: a string description of the condition

Extract_1d

The extract_1d step extracts a 1-d signal from a 2-d dataset and writes a spectrum to a product. The extraction information is contained in the JSON reference file.

Reference File Types

The reference file is a text file that uses JSON to hold the information needed.

CRDS Selection Criteria

The file is selected based on the values of DETECTOR and FILTER (and GRATING for NIRSpec).

Extract_1D Reference File 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 to use is found by matching the slit name in the science data with the value of key `id`.

The following keys are supported (but for IFU data, see below). Key `id` is required for any element of the `apertures` list that may be used; the value of `id` is compared with the slit name (except for a full-frame input image) to select the appropriate aperture. Key `dispaxis` is similarly required. Key `region_type` can be omitted, but if it is specified, its value must be "target". The source extraction region can be specified with `ystart`, `ystop`, etc., but a more flexible alternative is to use `src_coeff`. If background is to be subtracted, this should be specified by giving `bkg_coeff`. These are described in more detail below.

- id: the slit name, e.g. "S200A1" (string)
- dispaxis: dispersion direction, 1 for X, 2 for Y (int)
- xstart: first pixel in the horizontal direction, X (int)
- xstop: last pixel in the horizontal direction, X (int)
- ystart: first pixel in the vertical direction, Y (int)
- ystop: last pixel in the vertical direction, Y (int)
- 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_order: order of polynomial fit to background regions (int)
- extract_width: number of pixels in cross-dispersion direction (int)

If `src_coeff` is given, those coefficients take priority for specifying the source extraction region in the cross-dispersion direction. `xstart` and `xstop` (or `ystart` and `ystop` if `dispaxis` is 2) will still be used for the limits in the dispersion direction. Background subtraction will be done if and only if `bkg_coeff` is given. See below for further details.

For IFU cube data, these keys are used instead of the above:

- id: the slit name, but this can be "ANY" (string)
- x_center: X pixel coordinate of the target (pixels, float, the default is the center of the image along the X axis)
- y_center: Y pixel coordinate of the target (pixels, float, the default is the center of the image along the Y axis)
- radius: (only used for a point source) the radius of the circular extraction aperture (pixels, float, default is one quarter of the smaller of the image axis lengths)
- subtract_background: (only used for a point source) if true, subtract a background determined from an annulus with inner and outer radii given by `inner_bkg` and `outer_bkg` (boolean)
- inner_bkg: (only for a point source) radius of the inner edge of the background annulus (pixels, float, default = `radius`)
- outer_bkg: (only for a point source) radius of the outer edge of the background annulus (pixels, float, default = `inner_bkg * sqrt(2)`)
- width: (only for an extended source) the width of the rectangular extraction region; if `theta = 0`, the width side is along the X axis (pixels, float, default is half of the smaller image axis length)
- height: (only for an extended source) the height of the rectangular extraction region; if `theta = 0`, the height side is along the Y axis (pixels, float, default is half of the smaller image axis length)
- angle: (only for an extended source) the counterclockwise rotation angle of the `width` side from the positive X axis (degrees)
- 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 5)

The rest of this description pertains to the parameters for non-IFU data.

If `src_coeff` is not given, the extraction limits can be specified by `xstart`, `xstop`, `ystart`, `ystop`, and `extract_width`. Note that all of these values are integers. (It was intended that the start and stop limits be inclusive; the current code may not be consistent in this regard, but it will be so in the next release. To specify the cross-dispersion limits precisely, use `src_coeff`.) If `dispaxis` is 1, the zero-indexed limits in the dispersion direction are `xstart` and `xstop`; if `dispaxis` is 2, the dispersion limits are `ystart` and `ystop`. (The dispersion limits can be given even if `src_coeff` has been used for defining the cross-dispersion limits.) The limits in the cross-dispersion direction can be given by `ystart` and `ystop` (or `xstart` and `xstop` if `dispaxis` is 2). If `extract_width` is also given, that takes priority over `ystart` to `ystop` (for `dispaxis` = 1) for the extraction width, but `ystart` and `ystop` (for `dispaxis` = 1) will still be used to define the middle in the cross-dispersion direction. Any of these parameters can be modified by the step code if the extraction region would extend outside the input image, or outside the domain specified by the WCS.

The source extraction region can be specified more precisely by giving `src_coeff`, coefficients for polynomial functions for the lower and upper limits of the source extraction region. As described in the previous paragraph, using this key will override the values of `ystart` and `ystop` (if `dispaxis` is 1) or `xstart` and `xstop` (if `dispaxis` is 2), and `extract_width`. These polynomials are functions of either wavelength (in microns) or pixel number (pixels in the dispersion direction, with respect to the input 2-D slit image), specified by the key `independent_var`. The current default is "wavelength", but this may change to "pixel" in the future, so if the order of the polynomials for source or background is greater than zero, `independent_var` should be specified explicitly. The values of these polynomial functions are pixel numbers in the direction perpendicular to dispersion. More than one source extraction region may be specified, though this is not expected to be a typical case.

Background regions are specified by giving `bkg_coeff`, coefficients for polynomial functions for the lower and upper limits of one or more regions. Background subtraction will be done only if `bkg_coeff` is given in the reference file. See below for an example. See also `bkg_order` below.

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

If the dispersion direction is vertical, replace "lower" with "left" and "upper" with "right" in the above description.

Note 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 (this should not be confused with `bkg_order`, described below). The number of coefficients must be at least one, and there is no predefined upper limit. 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 image (identified by key `id`).

The background is determined independently for each column (or row, if `dispaxis` is 2) of the spectrum. The `smoothing_length` parameter is the width of a boxcar for smoothing the background in the dispersion direction. If this is not specified, either in the reference file, the config file, or on the command line, no smoothing will be done along the dispersion direction. Following background smoothing (if any), for each column (row), a polynomial of order `bkg_order` will be fit to the pixel values in that column (row) in all the background regions. If not specified, a value of 0 will be used, i.e. a constant function, the mean value. The polynomial will then be 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.

Extract_2d

The extract_2d step extracts a 2-D cutout for each spectrum in an exposure. It is saved as a "SCI" extension. It also computes and saves the wavelengths in a separate extension with EXTNAME "WAVELENGTH". It works on Nirspec MSA and fixed slits, as well as on NIRISS and NIRCAM slitless observations. Point source Nirspec wavelengths are (optionally) corrected for An optional wavelength zero-point correction is applied to Nirspec point source observations when the source is not centered in the slit. The data for the correction is saved in a WAVECORR reference file.

Reference File Types

The extract_2d step uses the following reference files:

reftype	description	Instrument
wavecorr	NIRSPEC wavelength zero-point correction	NIRSPEC

CRDS Selection Criteria For Each Reference File Type

WAVECORR

The WAVECORR reference file is selected based on EXP_TYPE of the science data. The reference file s relevant only for Nirspec observations with EXP_TYPE of NRS_FIXEDSLIT, NRS_MSASPEC, NRS_BRIGHTOBJ.

Reference File Formats For Each Reference File Type

WAVECORR

The WAVECORR file contains reference data about the NIRSPEC wavelength zero-point correction. The reference data is described in the NIRSPEC Technical Note ESA-JWST–SCI-NRS-TN-2016-018.

apertures

aperture_name Aperture name. :variance: Estimated variance on the zero-point offset. :width: Aperture width [SLIT] or pitch [MOS]. :zero_point_offset: Zero-point offset as a function of wavelength (in m)

and source offset within the aperture (in units of fraction of the aperture width [SLIT] or pitch [MOS]).

Flat field

The flat-field correction is applied by dividing both the science data and error images by the flat-field image in the reference file.

Reference File Types

There are four reference file types for the flat_field step. Reftype FLAT is used for all data except NIRSpec. NIRSpec data use three reftypes: FFLAT (fore optics), SFLAT (spectrograph optics), and DFLAT (detector).

CRDS Selection Criteria

For MIRI Imaging, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, FILTER, READPATT, and SUBARRAY in the science data file.

For MIRI MRS, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, BAND, READPATT, and SUBARRAY of the science data file.

For NIRCam, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, FILTER, and PUPIL of the science data file.

For NIRISS, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, and FILTER of the science data file.

For NIRSpec, flat-field reference files are selected based on the values of INSTRUME, DETECTOR, FILTER, GRATING, and EXP_TYPE of the science data file.

Reference File Formats for MIRI, NIRCAM, and NIRISS

Except for NIRSpec modes, flat-field reference files are FITS format with 3 IMAGE extensions and 1 BINTABLE extension. The primary data array is assumed to be empty. The 3 IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	2	ncols x nrows	float
ERR	2	ncols x nrows	float
DQ	2	ncols x nrows	integer

The BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array.

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 1032 x 1024 for MIRI, unless data were taken in subarray mode.

For slit spectroscopy, a set of SCI, ERR and DQ extensions can be provided for each aperture (identified by the detector subarray onto which the spectrum is projected).

A single DQ_DEF extension provides the data-quality definitions for all of the DQ arrays, which must use the same coding scheme. The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the value of 2^BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Reference File Formats for NIRSpec

For NIRSpec 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 there may be 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, as the allocated array size can be larger than is 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 file 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, 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 flat field 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 flat field contains four sets (one for each MSA quadrant) of images, WAVELENGTH tables, and FAST_VARIATION tables. The images are unique to the fore optics flat fields, however. The image "pixels" correspond to micro-shutter array slits, rather than to detector pixels. The array size is 365 rows by 171 columns, and there are multiple planes to handle the slow variation of flat field with wavelength.

For the spectrograph optics, 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 flat field 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.

Fore Optics (FFLAT)

There are 3 types of FFLAT reference files: fixed slit, msa spec, and IFU. For each type the primary data array is assumed to be empty.

Fixed Slit

The fixed slit references 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, number of the initial values of the wavelength and data arrays to use
- 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 references files have EXP_TYPE=NRS_MSASPEC, and contain data pertaining to each of the 4 quadrants. For each quadrant, there are 3 IMAGE extensions, a BINTABLE extension labeled WAVELENGTH, and a BINTABLE extension labeled FAST_VARIATION. The file also contains one BINTABLE extension labeled DQ_DEF.

The IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x nelem	float
ERR	3	ncols x nrows x nelem	float
DQ	3	ncols x nrows x nelem	integer

For all 3 of these extensions, the EXTVER keyword indicates the quadrant number, 1 to 4. Each plane of the SCI array gives the flat_field value for each aperture (slitlet) in the quadrant 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, number of the initial values of the wavelength and data arrays to use
- 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, as the same wavelength-dependent value is applied to all pixels in the quadrant.

The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the value of 2^BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

IFU

The IFU reference files have EXP_TYPE=NRS_IFU, a BINTABLE extension labeled FAST_VARIATION, and a BINTABLE extension labeled DQ_DEF.

The FAST_VARIATION table contains four columns:

- slit_name: the string "ANY"
- nelem: integer, number of the initial values of the wavelength and data arrays to use
- wavelength: float 1-D array, values of wavelength
- data: float 1-D array, flat field values for each wavelength

There is a single row in the table.

The DQ_DEF table contains the bit assignments used in the DQ arrays. The table contains the 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the value of 2^BIT
- NAME: the string mnemonic name of the data quality condition

- DESCRIPTION: a string description of the condition

Spectrograph (SFLAT)

There are 3 types of SFLAT reference files: fixed slit, msa spec, and IFU. For each type the primary data array is assumed to be empty.

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, number of the initial values of the wavelength and data arrays to use
- 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 references files have EXP_TYPE=NRS_MSASPEC. There are 3 IMAGE extensions, a BINTABLE extension labeled WAVELENGTH, a BINTABLE extension labeled FAST_VARIATION, and a BINTABLE extension labeled DQ_DEF.

The IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x n_wl	float
ERR	3	ncols x nrows x n_wl	float
DQ	3	ncols x nrows x n_wl	integer

The keyword NAXIS3 in these 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, number of the initial values of the wavelength and data arrays to use
- 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 determined by 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.

The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the value of 2^BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

IFU

The IFU reference files have EXP_TYPE=NRS_IFU, and has a BINTABLE extension labeled FAST_VARIATION, and a BINTABLE extension labeled DQ_DEF.

The IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows	float
ERR	3	ncols x nrows	float
DQ	3	ncols x nrows	integer

The FAST_VARIATION table contains four columns:

- slit_name: the string "ANY"
- nelem: integer, number of the initial values of the wavelength and data arrays to use
- 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, as the same wavelength-dependent value is applied to all pixels in the quadrant.

(Is this paragraph true - I copied it from above) 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 determined by 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.

The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the value of 2^BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Detector (DFLAT)

There is only one type of DFLAT reference file, and it contains 3 IMAGE extensions, a BINTABLE extension labeled WAVELENGTH, a BINTABLE extension labeled FAST_VARIATION, and a BINTABLE extension labeled DQ_DEF.

The IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	3	ncols x nrows x n_wl	float
ERR	3	ncols x nrows	float
DQ	3	ncols x nrows	integer

The keyword NAXIS3 in the SCI IMAGE 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, number of the initial values of the wavelength and data arrays to use
- 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, as the same wavelength-dependent value is applied to all pixels.

The DQ_DEF table contains the bit assignments used in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the value of 2^BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Fringe

This step applies a fringe correction to the SCI data of an input data set by dividing the SCI and ERR arrays by a fringe reference image. In particular, the SCI array from the fringe reference file is divided into the SCI and ERR arrays of the science data set. Only pixels that have valid values in the SCI array of the reference file will be corrected. This correction is applied only to MIRI MRS (IFU) mode exposures, which are always single full-frame 2-D images.

Reference File Types

The fringe correction step uses a FRINGE reference file, which has the same format as the FLAT reference file. This correction is applied only to MIRI MRS (IFU) mode exposures, which are always single full-frame 2-D images.

CRDS Selection Criteria

Fringe reference files are selected by DETECTOR and GRATNG14.

Reference File Format

Fringe reference files are FITS format with 3 IMAGE extensions and 1 BINTABLE extension. The primary data array is assumed to be empty. The 3 IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
SCI	2	ncols x nrows	float
ERR	2	ncols x nrows	float
DQ	2	ncols x nrows	integer

Image dimensions should be 1032 x 1024.

The BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array.

Jump Detection

The jump step 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.

The Jump Detection step uses two reference files: Gain and Readnoise. Both are necessary for proper computation of noise estimates.

The gain values are used to temporarily convert the pixel values from units of DN to electrons. It is assumed that the detector gain can vary from one pixel to another, so gain values are stored as 2-D images. The gain is given in units of electrons/DN.

It is assumed that the read noise can vary from pixel to pixel, so the read noise is also stored as a 2-D image. The values in the reference file are assumed to be per CDS pair of reads, as opposed to the read noise for a single read. The read noise is given in units of DN.

Reference File Types

The Jump step uses two reference files: GAIN and READNOISE. The gain values are used to temporarily convert the pixel values from units of DN to electrons. The read noise values are used as part of the noise estimate for each pixel. Both are necessary for proper computation of noise estimates.

CRDS Selection Criteria

GAIN Reference Files

The GAIN reference file is selected based on instrument, detector and, where necessary, subarray.

READNOISE Reference Files

The READNOISE reference file is selected by instrument, detector and, where necessary, subarray.

Reference File Formats

GAIN Reference Files

The gain reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of gain values (in e/DN) per pixel. The REFTYPE value is `GAIN`.

READNOISE Reference Files

The read noise reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of read noise values per pixel. The units of the read noise should be electrons and should be the CDS (Correlated Double Sampling) read noise, i.e. the effective noise between any pair of non-destructive detector reads. The REFTYPE value is `READNOISE`.

Linearity

The linearity correction corrects the integrated counts in the science images for the non-linear response of the detector. 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 values from the linearity reference file DQ array are propagated into the PIXELDQ array of the input science exposure using a bitwise OR operation.

Reference File Types

The linearity correction step uses a LINEARITY reference file.

CRDS Selection Criteria

Linearity reference files are selected by INSTRUME and DETECTOR.

Reference File Format

Linearity reference files are FITS format with 2 IMAGE extensions and 1 BINTABLE extension. The primary data array is assumed to be empty. The 2 IMAGE extensions have the following characteristics:

EXTNAME	NAXIS	Dimensions	Data type
COEFFS	3	ncols x nrows x ncoeffs	float
DQ	2	ncols x nrows	integer

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 BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array.

Path Loss

The pathloss reference file gives the path loss correction as a function of wavelength. There are two types of pathloss calibrations performed: for point sources and for uniform sources.

The point source entry in the reference file is a 3-d array with the pathloss correction as a function of wavelength and decenter within the aperture. The pathloss correction interpolates the 3-d array at the location of a point source to provide a 1-d array of pathloss vs. wavelength. This 1-d array is attached to the data model in the pathloss_pointsource attribute, with corresponding wavelength array in the wavelength_pointsource attribute.

The uniform source entry has a 1-d array of pathloss vs. wavelength. This array is attached to the data model in the pathloss_uniformsource attribute, along with the wavelength array in the wavelength_uniformsource attribute.

Reference File Types

The pathloss correction step uses a pathloss reference file.

CRDS Selection Criteria

Pathloss reference files are selected on the basis of EXP_TYPE values for the input science data set. Only NIRSPEC IFU, FIXEDSLIT and MSA data, and NIRISS SOSS data perform a pathloss correction.

Pathloss Reference File Formats

The PATHLOSS reference files are FITS files with extensions for each of the aperture types. The FITS primary data array is assumed to be empty.

The NIRSPEC IFU reference file has 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 NIRSPEC FIXEDSLIT reference file has this FITS structure:

HDU No.	Name	Type	Cards	Dimensions	Format
0	PRIMARY	PrimaryHDU	15	()	
1	PS	ImageHDU	29	(21, 21, 21)	float64
2	PSVAR	ImageHDU	29	(21, 21, 21)	float64
3	UNI	ImageHDU	19	(21,)	float64
4	UNIVAR	ImageHDU	19	(21,)	float64
5	PS	ImageHDU	29	(21, 21, 21)	float64
6	PSVAR	ImageHDU	29	(21, 21, 21)	float64
7	UNI	ImageHDU	19	(21,)	float64
8	UNIVAR	ImageHDU	19	(21,)	float64
9	PS	ImageHDU	29	(21, 21, 21)	float64
10	PSVAR	ImageHDU	29	(21, 21, 21)	float64
11	UNI	ImageHDU	19	(21,)	float64
12	UNIVAR	ImageHDU	19	(21,)	float64
13	PS	ImageHDU	29	(21, 21, 21)	float64
14	PSVAR	ImageHDU	29	(21, 21, 21)	float64
15	UNI	ImageHDU	19	(21,)	float64
16	UNIVAR	ImageHDU	19	(21,)	float64

HDU #1-4 are for the S200A1 aperture, while #5-8 are for S200A2, #9-12 are for S200B1 and #13-16 are for S1600A1. Currently there is no information for the S400A1 aperture.

The NIRSPEC IFU reference file just has 4 extensions after the primary HDU, as the behavior of each slice is considered identical.

The NIRSPEC MSASPEC reference file has 2 sets of 4 extensions, one for the 1x1 aperture size, and one for the 1x3 aperture size. Currently there are no other aperture sizes.

Photom

The photom step copies flux conversion information from the photometric reference table into the science product. The step searches the reference table for the row that matches the parameters of the exposure; the row contains a scalar conversion constant, as well as optional arrays of wavelength and relative response (as a function of wavelength). The scalar conversion constant is copied into the keyword PHOTMJSR in the primary header of the science product, and, if the wavelength and relative response arrays are populated in the selected row, those arrays are copied to a table extension called "RELSENS".

If the science data are from an imaging mode, the data from the pixel area map reference file will also be copied into the science data product. The 2-D data array from the pixel area map will be copied into an image extension called "AREA", and the values of the PIXAR_SR and PIXAR_A2 keywords in the photom reference table will also be copied into keywords of the same name in the primary header.

Reference File Types

The photom step uses a photom reference file and a pixel area map reference file. The pixel area map reference file is only used when processing imaging-mode observations.

CRDS Selection Criteria

PHOTOM Reference Files

For FGS, photom reference files are selected based on the values of INSTRUME and DETECTOR in the science data file.

For MIRI photom reference files are selected based on the values of INSTRUME and DETECTOR in the science data file.

For NIRCam, photom reference files are selected based on the values of INSTRUME and DETECTOR in the science data file.

For NIRISS, photom reference files are selected based on the values of INSTRUME and DETECTOR in the science data file.

For NIRSpec, photom reference files are selected based on the values of INSTRUME and EXP_TYPE in the science data file.

A row of data within the table that matches the mode of the science exposure is selected by the photom step based on criteria that are instrument mode dependent. The current row selection criteria are:

- FGS: No selection criteria (table contains a single row)
- **MIRI:**
  - Imager: Filter and Subarray
  - IFUs: Band
- NIRCam: Filter and Pupil
- NIRISS: Filter, Pupil, and Order number
- **NIRSpec:**
  - Fixed Slits: Filter, Grating, and Slit name
  - IFU and MSA: Filter and Grating

AREA map Reference Files

For FGS, photom reference files are selected based on the values of INSTRUME and DETECTOR in the science data file.

For MIRI photom reference files are selected based on the values of INSTRUME, DETECTOR, and EXP_TYPE in the science data file.

For NIRCam, photom reference files are selected based on the values of INSTRUME, DETECTOR, and EXP_TYPE in the science data file.

For NIRISS, photom reference files are selected based on the values of INSTRUME, DETECTOR, and EXP_TYPE in the science data file.

For NIRSpec, photom reference files are selected based on the values of INSTRUME, DETECTOR, and EXP_TYPE in the science data file.

Reference File Formats

PHOTOM Reference Files

Photom reference files are FITS format with a single BINTABLE extension. The primary data unit is always empty. The columns of the table vary with instrument according to the selection criteria listed above. The first few columns always correspond to the selection criteria, such as Filter and Pupil, or Filter and Grating. The remaining columns contain the data relevant to the photometric conversion and consist of PHOTMJSR, UNCERTAINTY, NELEM, WAVELENGTH, and RELRESPONSE.

- FILTER (string) - MIRI, NIRCam, NIRISS, NIRSpec
- PUPIL (string) - NIRCam, NIRISS
- ORDER (integer) - NIRISS
- GRATING (string) - NIRSpec
- SLIT (string) - NIRSpec Fixed-Slit
- SUBARRAY (string) - MIRI Imager/LRS
- BAND (string) - MIRI MRS
- PHOTMJSR (float) - all instruments
- UNCERTAINTY (float) - all instruments
- NELEM (int) - if NELEM > 0, then NELEM entries are read from each of the WAVELENGTH and RELRESPONSE arrays
- WAVELENGTH (float 1-D array)
- RELRESPONSE (float 1-D array)

The primary header of the photom reference file contains the keywords PIXAR_SR and PIXAR_A2, which give the average pixel area in units of steradians and square arcseconds, respectively.

AREA Reference Files

Pixel area map reference files are FITS format with a single image extension with 'EXTNAME=SCI', which contains a 2-D floating-point array of values. The FITS primary data array is always empty. The primary header contains the keywords PIXAR_SR and PIXAR_A2, which should have the same values as the keywords in the header of the corresponding photom reference file.

Constructing a PHOTOM Reference File

The most straight-forward way to construct a PHOTOM reference file is to populate a photom data model within python and then save the data model to a FITS file. Each instrument has its own photom data model, which contains the columns of information unique to that instrument:

- NircamPhotomModel
- NirissPhotomModel
- NirspecPhotomModel
- MiriImgPhotomModel
- MiriMrsPhotomModel

A NIRISS photom reference file, for example, could be constructed as follows from within the python environment:

```
>>> from jwst import models
>>> import numpy as np
>>> output=models.NirissPhotomModel()
>>> filter=np.array(['F277W','F356W','CLEAR'])
>>> pupil=np.array(['CLEARP','CLEARP','F090W'])
>>> photf=np.array([1.e-15,2.e-15,3.e-15])
>>> uncer=np.array([1.e-17,2.e-17,3.e-17])
>>> nelem=np.zeros(3)
>>> wave=np.zeros(3)
>>> resp=np.zeros(3)
>>> data=np.array(zip(filter,pupil,photf,uncer,nelem,wave,resp),dtype=output.phot_
↪table.dtype)
>>> output.phot_table=data
>>> output.save('niriss_photom_0001.fits')
```

Ramp Fitting

The Ramp Fitting step uses two reference files: Gain and Readnoise. Both are necessary for proper computation of noise estimates.

The gain values are used to temporarily convert the pixel values from units of DN to electrons. It is assumed that the detector gain can vary from one pixel to another, so gain values are stored as 2-D images. The gain is given in units of electrons/DN.

It is assumed that the read noise can vary from pixel to pixel, so the read noise is also stored as a 2-D image. The values in the reference file are assumed to be per CDS pair of reads, as opposed to the read noise for a single read. The read noise is given in units of DN.

Reference File Types

The Ramp Fitting step uses two reference files: GAIN and READNOISE. 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 read noise values are used as part of the noise estimate for each pixel. Both are necessary for proper computation of noise estimates.

CRDS Selection Criteria

GAIN Reference Files

The GAIN reference file is selected based on instrument, detector and, where necessary, subarray.

READNOISE Reference Files

The READNOISE reference file is selected by instrument, detector and, where necessary, subarray.

Reference File Formats

GAIN Reference Files

The gain reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of gain values (in e/DN) per pixel. The REFTYPE value is `GAIN`.

READNOISE Reference Files

The read noise reference file is a FITS file with a single IMAGE extension, with `EXTNAME=SCI`, which contains a 2-D floating-point array of read noise values per pixel. The units of the read noise should be electrons and should be the CDS (Correlated Double Sampling) read noise, i.e. the effective noise between any pair of non-destructive detector reads. The REFTYPE value is `READNOISE`.

Refpix

The refpix step corrects for bias drift. The reference-pixel reference file contains frequency-dependent weights that are used to compute (in Fourier space) the filtered reference pixels and reference output for the reference-pixel correction scheme that is applied to NIRSpec data when exposures use the IRS2 readout pattern. Only the NIRSpec IRS2 readout format requires a reference file; no other instruments or exposure modes require a reference file for this step.

For each sector, the correction is applied as follows: data * alpha[i] + reference_output * beta[i]. `Alpha` and `beta` are 2-D arrays of values read from the reference file. The first axis is the sector number (but only for the normal pixel data and reference pixels, not the reference output). The second axis has length 2048 * 712, corresponding to the time-ordered arrangement of the data. `Data` is the science data for the current integration. The shape is expected to be (ngroups, ny, 3200), where ngroups is the number of groups, and ny is the pixel height of the image. The width 3200 of the image includes the "normal" pixel data, plus the embedded reference pixels, and the reference output. `Reference_output` is the length of the reference output section.

Reference File Types

The refpix step only uses the refpix reference file 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.

CRDS Selection Criteria

Refpix reference files are selected by DETECTOR and READPATT.

Reference File Format

A single IRS2 extension provides the complex coefficients for the correction, and contains 8 columns ALPHA_0, ALPHA_1, ALPHA_2, ALPHA_3, BETA_0, BETA_1, BETA_2, and BETA_3. The ALPHA arrays contains correction multipliers to the data, and the BETA arrays contains correction multiplier to the reference output. Both arrays have 4 components - one for each sector.

RSCD

This step performs an RSCD (Reset Switch Charge Decay) correction by adding a function of time, frame by frame, to a copy of the input science. The reference file contains a table with the parameters of the function.

Reference File Types

The RSCD correction step uses an RSCD reference file. This correction only applied to integrations > 1. The correction to be added to the input data has the form:

```
corrected = input + dn_accumulated * scale * exp(-T / tau)
```

where dn_accumulated is the DN level that was accumulated for the pixel from the previous integration.

CRDS Selection Criteria

RSCD reference files are selected on the basis of INSTRUME and DETECTOR values for the input science data set. The reference file for each detector is a table of values based on READPATT (FAST, SLOW), SUBARRAY (FULL or one the various subarray types), and ROWS type (even or odd row). The correction values, tau and scale, are read in separately for even and odd rows, based on the readout pattern and if it is for the full array or one of the imager subarrays. The table actually contains the parameters for a double-exponential function, but currently only the single exponential values are used.

RSCD Reference File Format

The RSCD reference files are FITS files with a BINTABLE extension. The FITS primary data array is assumed to be empty.

The BINTABLE extension contains the row-selection criteria (SUBARRAY, READPATT, and ROW type) and the parameters for a double-exponential correction function. It uses `EXTNAME=RSCD` and contains seven columns:

- SUBARRAY: string, FULL or a subarray name
- READPATT: string, SLOW or FAST
- ROWS: string, EVEN or ODD
- TAU1: float, e-folding time scale for the first exponential (unit is frames)
- SCALE1: float, scale factor for the first exponential
- TAU2: float, e-folding time scale for the second exponential (frames)
- SCALE2: float, scale factor for the second exponential

Saturation

The saturation level of each pixel (in units of DN) is stored as a 2-D image in the reference file. For each group in the science data file, the pipeline compares each pixel's DN value with its saturation level. If the pixel exceeds the saturation level, then the SATURATED flag is set for that pixel in the corresponding plane of the GROUPDQ array – and in all subsequent planes. No saturation check is performed on pixels for which the flag NO_SAT_CHECK is set.

Reference File Types

The saturation step uses a SATURATION reference file.

CRDS Selection Criteria

Saturation reference files are selected on the basis of INSTRUME, DETECTOR, and SUBARRAY values from the input science data set.

SATURATION Reference File Format

Saturation reference files are FITS format with with 2 IMAGE extensions: `SCI` and `DQ`, which are both 2-D integer arrays, and 1 BINTABLE extension.

The values in the SCI array give the saturation threshold in units of DN for each pixel. The saturation reference file also contains a `DQ_DEF` table extension, which lists the bit assignments for the flag conditions used in the DQ array.

The BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Straylight

The stray-light step is applied to MIRI MRS data only, in which case a stray-light MASK reference file is used to designate which pixels are science pixels and which pixels fall in-between the slices. Each illuminated pixel on the array has a signal that is the sum of direct illumination and the scattering from neighboring areas. Only the pixels located between the slices are areas of indirect illumination. The illumination on the inter-slice pixels are used to determine a stray-light component to subtract from each science pixel.

Reference File Types

The MIRI MRS straylight correction step uses a straylight mask. There are three MIRI MRS SW masks, one for each of the three bands (SHORT,MEDIUM and LONG).

CRDS Selection Criteria

MIRI MRS reference files are selected on the basis of INSTRUME, DETECTOR, and BAND values from the input science data set.

MIRI MRS straylight Reference File Format

The straylight mask reference files are FITS files with one IMAGE extension. This image extension is a 2-D integer mask file of size 1032 X 1024. The mask contains values of 1 for pixels that fall in the slice gaps and values of 0 for science pixels. The straylight algorithm only uses pixels that fall in the slice gaps to determine

Superbias

The superbias subtraction step removes the fixed detector bias from a science data set by subtracting a superbias reference image. 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 bit-wise OR operation.

Reference File Types

The superbias subtraction step uses a SUPERBIAS reference file.

CRDS Selection Criteria

Superbias reference files are selected on the basis of the INSTRUME, DETECTOR, READPATT and SUBARRAY values of the input science data set.

SUPERBIAS Reference File Format

Superbias 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	2	ncols x nrows	float
ERR	2	ncols x nrows	float
DQ	2	ncols x nrows	integer

The BINTABLE extension contains the bit assignments used in the DQ array. It uses `EXTNAME=DQ_DEF` and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

12.1.40 Reference Pixel Correction

Description

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 a 4 columns (1 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 ramp, and it should contain both a science ('SCI') extension and a data quality ('DQ') extension. The latter extension is normally added by the dq_init step, so running this step is a prerequisite for the refpix step.

Algorithm

The algorithm for the NIR and MIR detectors is different.

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` runtime parameter, as 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.
    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 runtime 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 runtime parameter `side_gain`, which defaults to 1.0) is subtracted from the full group on a row-by-row basis.
4. Transform the data back to the JWST focal plane, or DMS, frame.

MIR Detector Data

1. MIR data is already 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` runtime parameter, as 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.
    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'.

Subarrays

Subarrays are treated slightly differently. Once again, the data are flipped and/or rotated to convert to the detector frame

NIR Data

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 dq 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 this step before running the refpix step on subarray data.

If the science dataset has at least 1 group with no valid reference pixels, the refpix step is skipped and the S_REFPIX header keyword is set to 'SKIPPED'.

MIR Data

The refpix correction is skipped for MIRI subarray data.

Reference File Types

The refpix step only uses the refpix reference file 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.

CRDS Selection Criteria

Refpix reference files are selected by DETECTOR and READPATT.

Reference File Format

A single extension, with a EXTNAME keyword of 'IRS2', provides the complex coefficients for the correction, and contains 8 columns ALPHA_0, ALPHA_1, ALPHA_2, ALPHA_3, BETA_0, BETA_1, BETA_2, and BETA_3. The ALPHA arrays contains correction multipliers to the data, and the BETA arrays contains correction multiplier to the reference output. Both arrays have 4 components - one for each sector.

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.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

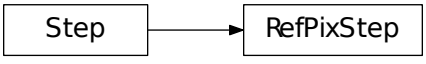
reference_file_types = ['refpix']

spec = '\n odd_even_columns = boolean(default=True)\n use_side_ref_pixels = boolean(de

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



12.1.41 Resample

Description

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 latest assign_wcs pipeline step.

The resample step can take as input either

- a single 2D input image updated by assign_wcs
- an association table (in json format)

The pipeline 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. This 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.

It 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 define output product. This mapping function gets passed to cdriz to drive the actual drizzling to create the output product.

A full description of the drizzling algorithm, and parameters for drizzling, can be found in the DrizzlePac Handbook (http://drizzlepac.stsci.edu).

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: *jwst.stpipe.Step*

Resample input data onto a regular grid using the drizzle algorithm.

Parameters **input** (*DataModel 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.
- ****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

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.

Attributes Documentation

reference_file_types = ['drizpars']

spec = "\n pixfrac = float(default=None)\n kernel = string(default=None)\n fillval = s

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:

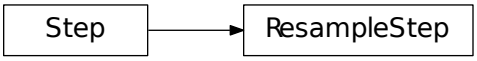
pixfrac = float(default=None) kernel = string(default=None) fillval = string(default=None) weight_type = option('exptime', default=None)

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 overwerite 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.

Class Inheritance Diagram



Python Interface to Drizzle: ResampleData()

jwst.resample.resample Module

Classes

ResampleData(input_models[, output])	This is the controlling routine for the resampling process.

ResampleData

class jwst.resample.resample.**ResampleData**(*input_models*, *output=None*, ***pars*)
Bases: object (https://docs.python.org/3/library/functions.html#object)

This is the controlling routine for the resampling process. It loads and sets the various input data and parameters needed by the drizzle function and then calls the C-based cdriz.tdriz function to do the actual resampling.

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. Initializes all output arrays, including WHT and CTX arrays.
4. Passes all information for each input chip to drizzle function.
5. Updates output data model with output arrays from drizzle, including
   (eventually) a record of metadata from all input models.
```

Parameters

- **input_models** (*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

Methods Summary

blend_output_metadata(output_model)	Create new output metadata based on blending all input metadata.
do_drizzle()	Perform drizzling operation on input images's to create a new output
update_driz_outputs()	Define output arrays for use with drizzle operations.
update_fits_wcs(model)	Update FITS WCS keywords of the resampled image.

Methods Documentation

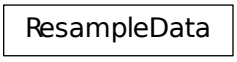
blend_output_metadata(*output_model*)
Create new output metadata based on blending all input metadata.

do_drizzle()
Perform drizzling operation on input images's to create a new output

update_driz_outputs()
Define output arrays for use with drizzle operations.

update_fits_wcs(*model*)
Update FITS WCS keywords of the resampled image.

Class Inheritance Diagram



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: *jwst.stpipe.Step*

Resample input data onto a regular grid using the drizzle algorithm.

Parameters **input** (*DataModel or Association*) – Single filename for either a single im-

age 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.
- ****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

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.

Attributes Documentation

reference_file_types = ['drizpars']

spec = "\n pixfrac = float(default=None)\n kernel = string(default=None)\n fillval = s

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) weight_type = option('exptime', 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 overwerite 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.

ResampleSpecStep

class jwst.resample.**ResampleSpecStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.resample.resample_step.ResampleStep*

ResampleSpecStep: Resample input data onto a regular grid using the drizzle algorithm.

Parameters **input** (MultSlitModel, *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.
- ****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.

Methods Summary

process(input)	This is where real work happens.

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

12.1.42 Reset Correction

Description

Assumptions

The reset correction is currently only implemented for MIRI data. It is assumed that the input science data have *NOT* had the zero group (or bias) subtracted. We also do not want the reset correction to remove the bias signal from the science exposure, therefore the reset correction for the first group is defined to be zero.

Background

Currently this step is only implemented for MIRI data. For MIRI data the initial groups in an integration suffer from two effects related to the resetting of the detectors. The first effect is that the first few samples starting an integration after a reset do not fall on the expected linear accumulation of signal. The most significant deviations ocurr 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 that appears on in these initial groups, before fading out by later groups.

The time constant associated with the reset anomaly is roughly a minute so for full array data the effect has faded out by ~group 20. On subarray data, where the read time depends on the size of the subarray, the reset anomaly affects more groups in an integration.

For multiple integration data the reset anomaly also varies in amplitude for the first set of integrations before settling down to a relatively constant correction for integrations greater than four 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. The current implementation uses a reset anomaly reference file for full array data containing a correction for the first 30 groups for integrations 1-4. The reference file was determined so that the correction is forced to be zero on the last group for each integration. 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 intergration 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.

CRDS Selection Criteria

Reset reference files are selected on the basis of INSTRUME, DETECTOR, READPATT and SUBARRAY values for the input science data set.

RESET 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 BINTABLE extension contains the bit assignments used in the DQ array. It uses `EXTNAME=DQ_DEF` and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

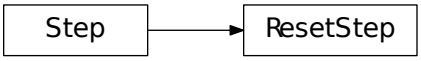
reference_file_types = ['reset']

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



12.1.43 Reset Switch Charge Decay (RSCD) Correction

Description

Assumptions

This correction is currently only implemented for MIRI data and is only for 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.

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 which 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 reset offsets and nonlinearities at the start of the integration. 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. The Reset Switch Charge Decay (RSCD) step corrects for the slow adjustment of the FET output to its asymptotic level after a reset. This correction is made for integrations > 1 and is based on the signal level in the last frame of the previous integration in the exposure. Between exposures the MIRI detectors are continually reset; however for a multiple integration exposure there is a single reset between integrations. The reset switch charge decay has an e-folding time scale ~ 1.3 * frame time. The affects 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 readout 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.

For MIRI multiple integration data, the reset switch decay causes the the initial groups in integrations after the first one to be offset from their expected linear accumulation of signal. The most significant deviations occur in groups 1 and 2. The amplitude of the difference between the expected value and the measured value varies for even and odd rows and is related to the signal in the last frame of the last integration.

The MIRI reset electronics also cause a zero-point offset in multiple integration data. Subsequent integrations after the first integration start at a lower DN level. The amplitude of this offset is proportional to the signal level in the previous integration. Fortunately this offset is constant for all the groups in the integration, thus has no impact on the slopes determined for each integration.

Algorithm

This correction is only applied to integrations > 1. The RSCD correction step applies an exponential decay correction based on coefficients in the reset switch charge decay reference file. The reference files are selected based on READOUT pattern (FAST or SLOW) and Subarray type (FULL or one of the MIRI defined subarray types). The reference file contains the information necessary to derive the scale factor and decay time to correct for the reset effects. The correction differs for even and odd row numbers.

The correction to be added to the input data has the form:

```
corrected data = input data 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 and dn_accumulated is the DN level that was accumulated for the pixel from the previous integration. Because of the last frame effect the value of the last group in an integration is not measured accurately. Therefore, the accumulated DN of the pixel from the previous integration (last group value) is estimated by extrapolating the ramp using the second to last and third to last groups.

In the case where the previous integration does not saturate the $scale$ term in Equation 1 is determined as follows:

$$scale = b1 * [Counts2^{b2} * [1/exp(Counts2/b3) - 1] \;\; Equation\ 2$$

The terms $b2$ and $b3$ are read in from the RSCD reference file. The following two additional equations are needed to calculate the $b1$ and $Counts2$ terms:

$$b1 \;=\; ascale * (illum_{zpt} + illum_{slope} * N + illum2 * N^2) \quad (Equation\ \ 2.1)\ Counts2 \;=\; Final\ DN\ in\ the\ last\ group\ in\ the\ last\ integration\ - Crossover\ Point\ \ (Equation\ 2.2)$$

In equation 2.1, N is the number of groups per integration and $ascale$, $illum_{zpt}$, $illum_{slope}$, and $illum2$ are read in from the RSCD reference file. The $Crossover\ Point$ in equation 2.2 is also read in from the RSCD reference file.

If the previous integration saturates, the $scale$ term in Equation 1 is found in the following manner:

$$scale_{\text{sat}} = slope * Counts3 + sat_{\text{mzp}} \;\; (Equation\ 3)$$

where $Counts3$ is an estimate of what the last group in the previous integration would have been if saturation did not exist. The $slope$ in equation 3 is calculated according to the formula:

$$slope = sat_{zp} + sat_{slope} * N + sat_2 * N^2 + evenrow_{corrections} \;\; (Equation 3.1).$$

The terms sat_{mzp}, sat_{zp}, sat_2, $evenrow_{corrections}$ are read in from the reference file.

All fourteen parameters tau, $b1$, $b2$, $b3$, $illum_{zpt}$, $illum_{slope}$, $illum2$, $CrossoverPoint$, sat_{zp}, sat_{slope}, sat_2, sat_{scale}, sat_{mzp}, and $evenrow_{corrections}$ are found in the RSCD reference files. There is a seperate set for even and odd rows for each READOUT mode and SUBARRAY type.

Subarrays

Currently the RSCD correction for subarray data is the same as it is for full array data. However, we anticipate a separate set of correction coefficients in the future.

Reference File Types

The RSCD correction step uses an RSCD reference file.

CRDS Selection Criteria

RSCD reference files are selected on the basis of INSTRUME and DETECTOR values for the input science data set. The reference file for each detector is a table of values based on READPATT (FAST, SLOW) , SUBARRAY (FULL or one the various subarray types) , and ROWS type (even or odd row). The fourteen correction values are read in separately for even and odd rows for the readout pattern and if it is for the full array or one of the imager subarrays.

RSCD Reference File Format

The RSCD reference files are FITS files with a BINTABLE extension. The FITS primary data array is assumed to be empty.

The BINTABLE extension contains the row-selection criteria (SUBARRAY, READPATT, and ROW type) and the parameters for a double-exponential correction function. It uses `EXTNAME=RSCD` and contains seventeen columns which are used in determining the correction for the equations given after the table.

- SUBARRAY: string, FULL or a subarray name
- READPATT: string, SLOW or FAST
- ROWS: string, EVEN or ODD
- TAU: float, e-folding time scale for the first exponential (unit is frames)
- ASCALE: float, b1 in equation
- POW: float, b2 in equation
- ILLUM_ZP: float
- ILLUM_SLOPE: float
- ILLUM2: float
- PARAM3: $b3$ in equation
- CROSSOPT: float, crossover point
- SAT_ZP: float
- SAT_SLOPE: float
- SAT2: float
- SAT_MZP: float
- SAT_ROWTERM: float
- SAT_SCALE: float

In order to explain where these parameters are used in the correction we will go over the correction equations given in the Description Section.

The general form of the correction to be added to the input data is:

```
corrected data = input data 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 and dn_accumulated is the DN level that was accumulated for the pixel from the previous integration. In case where the last integration does not saturate the $scale$ term in equation 1 is determined according to the equation:

$$scale = b1 * [Counts2^{b2} * [1/exp(Counts2/b3) - 1] \ \ (Equation\ 2)$$

The following two additional equations are used in Equation 2:

$b1 = ascale * (illum_{zpt} + illum_{slope} * N + illum2 * N^2)$ $(Equation\ 2.1)$:math:‘Counts{2} = Final , DN , in , the , last , group , in ; the , last , integration

, - Crossover , Point ; ; (Equation ; 2.2)‘

The parameters for equations 2, 2.1, and 2,2 are:

- $b2$ in equation 2 is table column POW from RSCD table
- $b3$ in equation 2 is table column PARAM3 from the RSCD table
- ascale in equation 2.1 is in the RSCD table
- $illum_{zpt}$ in equation 2.1 is in the RSCD table
- $illum_{slope}$ in equation 2.1 is in the RSCD table
- $illum2$ in equation 2.1 is in the RSCD table
- N in equation 2.1 is the number of groups per integration
- Crossover Point in equation 2.2 is CROSSOPT in the RSCD table

If the previous integration saturates, $scale$ is no longer calculated using equation 2 - 2.2, instead it is calculated using equations 3 and 3.1.

$scale_{\text{sat}} = slope * Counts3 + sat_{\text{mzp}}$ $(Equation\ 3)$

$slope = sat_{zp} + sat_{slope} * N + sat_2 * N^2 + evenrow_{corrections}$ $(Equation\ 3.1)$.

The parameters in equation 3 and 3.1 are:

- $Counts3$ in equation 3 is an estimate of the what the last group in the previous integration would have been if saturation did not exist
- sat_{mzp} in equation 3 is in the RSCD table
- $scale_{\text{sat}}$ in equation 3 is SAT_SCALE in the RSCD table
- sat_{zp} in equation 3.1 is in the RSCD table
- sat_{slope} in equation 3.1 is in the RSCD table
- sat_2 in equation 3.1 is SAT2 in the RSCD table
- $evenrow_{corrections}$ in equation 3.1 is SAT_ROWTERM in the RSCD table
- N is the number of groups per integration

Step Arguments

The RSCD correction has no step-specific arguments.

jwst.rscd Package

Classes

RSCD_Step([name, parent, config_file, ...])	RSCD_Step: Performs an RSCD correction to MIRI data by adding a function of time, frame by frame, to a copy of the input science data model.

RSCD_Step

class jwst.rscd.**RSCD_Step**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *jwst.stpipe.Step*

RSCD_Step: Performs an RSCD correction to MIRI data by adding a function of time, frame by frame, to a copy of 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.
- ****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

reference_file_types

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

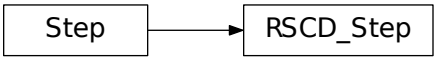
reference_file_types = ['rscd']

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



12.1.44 Saturation Detection

Description

The `saturation` step flags saturated pixel values. It loops over all integrations within an exposure, examining them group-by-group, comparing the science exposure values with defined saturation thresholds for each pixel. When it finds a pixel value in a given group that is above the threshold, it sets the `SATURATED` flag in the corresponding location of the GROUPDQ array in the science exposure.

Reference Files

This step requires a SATURATION reference file, which is used to specify the saturation threshold for each pixel. The saturation files are FITS format, with 2 IMAGE extensions: `SCI` and `DQ`. They are both 2-D integer arrays. The values in the SCI array give the saturation threshold in units of DN for each pixel. The saturation reference file also contains a `DQ_DEF` table extension, which lists the bit assignments for the flag conditions used in the DQ array.

For pixels having a saturation threshold set to NaN in the reference file, those thresholds will be replaced by 100000, a very high value that exceeds any possible science data pixel value. This ensures that these pixels will not be flagged by this step as saturated. The associated groupdq values will be flagged as NO_SAT_CHECK in the step output. Similarly, for pixels flagged as NO_SAT_CHECK in the reference file, they will be added to the dq mask, and have their saturation values set to be so high they will not be flagged as saturated.

The saturation reference files are selected based on instrument, detector and, where necessary, subarray.

Subarrays

The step will accept either full-frame or subarray saturation reference files. If only a full-frame reference file is available, the step will extract subarrays to match those of the science exposure. Otherwise, subarray-specific saturation reference files will be used if they are available.

Reference File Types

The saturation step uses a SATURATION reference file.

CRDS Selection Criteria

Saturation reference files are selected on the basis of INSTRUME, DETECTOR, and SUBARRAY values from the input science data set.

SATURATION Reference File Format

Saturation reference files are FITS format with with 2 IMAGE extensions: `SCI` and `DQ`, which are both 2-D integer arrays, and 1 BINTABLE extension.

The values in the SCI array give the saturation threshold in units of DN for each pixel. The saturation reference file also contains a `DQ_DEF` table extension, which lists the bit assignments for the flag conditions used in the DQ array.

The BINTABLE extension uses `EXTNAME=DQ_DEF` and contains the bit assignments of the conditions flagged in the DQ array, and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types

Methods Summary

process(input)	This is where real work happens.

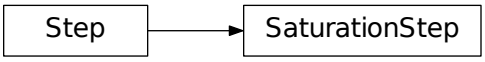
Attributes Documentation

reference_file_types = ['saturation']

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



12.1.45 SkyMatch

Description

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 combining multiple images into a mosaic. When running the `skymatch` step in a matching mode, it compares *total* signal levels in *the overlap regions* (instead of doing this comparison on a per-pixel basis, cf. *mrs_imatch step*) of a set of input images and computes the signal offsets for each image that will minimize the residuals across the entire set in the least squares sence. This comparison is performed directly on 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. By default the sky value computed for each image is recorded, but not actually subtracted from the images. Also note that the meaning of "sky background" depends on the chosen sky computation method.

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.

Algorithm

The `skymatch` step provides several methods for constant sky background value computations.

The first method, called `localmin`, essentially is an enhanced version of the original sky subtraction method used in older astrodrizzle (https://drizzlepac.readthedocs.io/en/latest/astrodrizzle.html) versions. 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 and user-supplied masks to remove "bad" pixels from being used for sky statistics computations. Values different from zero in user-supplied masks indicate "good" data pixels.

In addition to the classical `localmin` method, two other methods have been introduced: `globalmin` and `match`, as well as a combination of the two – `globalmin+match`.

- The `globalmin` method computes the minimum sky value across *all* input images. The resulting *single sky value* is then considered to be the background in *all input images*.
- The `match` algorithm computes constant (within an image) value corrections to be applied to each input image such that the mismatch in computed backgrounds between *all* pairs of images is minimized in the least squares sense. For each pair of images, the background 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 (at least) pair-wise intersection of images without having a common intersection region (on the sky) in all images.
- The `globalmin+match` algorithm combines the `match` and `globalmin` methods. It uses the `globalmin` algorithm to find a baseline sky value common to all input images and the `match` algorithm to "equalize" sky values among images.

In methods that find sky background levels in each image (`localmin`) or a single level for all images (`globalmin`), image statistics are usually computed using sigma clipping. If the input images contain vast swaths of empty sky, then the sigma clipping algorithm should be able to automatically exclude (clip) contributions from bright compact sources. In this case the measured "sky background" is the measured signal level from the "empty sky". On the other hand, the `match` method compares the *total* signal levels integrated over those regions in the images that correspond to common ("overlap") regions on the celestial sphere for both images being compared (comparison is pair-wise). This method is often used when there are no large "empty sky" regions in the images, such as when a large nebula occupies most of the view. This method cannot measure "true background", but rather additive corrections that need to be applied to the input images so that the total signal from the same part of the sky is equal in all images.

Step Arguments

The `skymatch` step has the following optional arguments:

General sky matching parameters:

- `skymethod` (str): The sky computation algorithm to be used. Allowed values: {`local`, `global`, `match`, `global+match`} (Default = `global+match`)
- `match_down` (boolean): 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)). (Default = `True` (https://docs.python.org/3/library/constants.html#True))

  Note: This setting applies *only* when `skymethod` is either `match` or `global+match`.

- `subtract` (boolean): Specifies whether the computed sky background values are to be subtracted from the images. (Default = `False` (https://docs.python.org/3/library/constants.html#False))

Image bounding polygon parameters:

- `stepsize` (int): Spacing between vertices of the images 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.

Sky statistics parameters:

- `skystat` (str): Statistic to be used for sky background value computations. Supported values are: 'mean', 'mode', 'midpt', and 'median'. (Default = 'mode')
- `dqbits` (str): Integer sum of all the DQ bit values from the input images DQ arrays that should be considered "good" when building masks for sky computations. 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 sky computations, then `dqbits` 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 setting `dqbits` to 12.

  Note:

  - The default value (0) will make *all* non-zero pixels in the DQ mask be considered "bad" pixels and the corresponding image pixels will not be used for sky computations.
  - Set `dqbits` to `None` (https://docs.python.org/3/library/constants.html#None) to turn off the use of image's DQ array for sky computations.
  - In order to reverse the meaning of the `dqbits` parameter 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 sky computations and to consider as "good" all other pixels (regardless of their DQ flag), set `dqbits` to `~4+8`, or `~4,8`. A `dqbits` string value of `~0` would be equivalent to setting `dqbits=None`.

- `lower` (float): 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. (Default = `None` (https://docs.python.org/3/library/constants.html#None))
- `upper` (float): 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. (Default = `None` (https://docs.python.org/3/library/constants.html#None))
- `nclip` (int): A non-negative number of clipping iterations to use when computing the sky value. (Default = 5)
- `lsig` (float): Lower clipping limit, in sigma, used when computing the sky value. (Default = 4.0)
- `usig` (float): Upper clipping limit, in sigma, used when computing the sky value. (Default = 4.0)
- `binwidth` (float): 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`. (Default = 0.1)

Limitations and Discussions

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 "sky values" in input images in their overlap regions (that

is common to a pair of 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 images "true" background may not be present at all and the measured sky may be the surface brightness of a large galaxy, nebula, etc.

Here is a brief list of possible limitations/factors that can affect the outcome of the matching (sky subtraction in general) algorithm:

- Because 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 the `upper` parameter to limit the use of bright objects in the sky computations.
- 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.
- 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.
- 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.

Reference Files

This step does not require any reference files.

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 pipeline step for sky matching.

Authors Mihai Cara

class jwst.skymatch.skymatch_step.**SkyMatchStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

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.
- ****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.

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.

reference_file_types = []

spec = '\n # General sky matching parameters:\n skymethod = option(\'local\', \'global

skymatch

The `skymatch` function performs the actual sky matching operations on the input image data models.

A module that provides functions for matching sky in overlapping images.

Authors Mihai Cara

`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 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 opposite to extended, diffuse sources).

  - **'global'**: compute a common sky value for all input image and groups of images. In this setting *match* will compute sky values for each input image/group, find the minimum sky value, and then it will set (and/or subtract) sky value of each input image to this minimum value. This method *may* be useful when 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 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 "measured" sky values for median image generation and CR rejection.

– **'global+match'**: first use **'match'** method to equalize sky values between images and then find a minimum "global" sky value in 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 `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 indipendently 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 is computed only in the part of the image that intersects other images.

  This makes `'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: Current algorithm is not capable of detecting cases when some subsets of intersecting images (from the input list of images) do not intersect at all 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 `'match'` and `'global'` methods in order to overcome the limitation of the `'match'` method described in the note above: it uses `'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 of input images.

  Initially, the footprint of a chip in an image is aproximated by a 2D planar rectangle representing the borders of chip's distorted image. After applying distortion model to this rectangle and progecting 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

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)

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*)

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://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.nd – A 2D array of image data.
- **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://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.nda – A 2D array that indicates what 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 prefereble that *id* be of a type with nice string representation.
- **skystat** (*callable*, *None* (https://docs.python.org/3/library/constants.html#None), *optional*) – A callable object that takes a either a 2D image (2D `numpy.ndarray` (https://docs.scipy.org/doc/numpy/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* (https://docs.python.org/3/library/constants.html#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* (https://docs.python.org/3/library/constants.html#None), *optional*) – A dictionary of various items to be stored within the *SkyImage* object.

calc_bounding_polygon(*stepsize=None*)

Compute image's bounding polygon.

Parameters **stepsize** (*int* (https://docs.python.org/3/library/functions.html#int), *None* (https://docs.python.org/3/library/constants.html#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 of tuples*, *None* (https://docs.python.org/3/library/constants.html#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.

id

Set or get *SkyImage*'s *id*.

While *id* can be of any type, it is prefereble that *id* be of a type with nice string representation.

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

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 verteces of the bounding polygon as a ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) of shape (N, 2) where N is the number of verteces + 1.

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.

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://docs.scipy.org/doc/numpy/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.

class jwst.skymatch.skyimage.**SkyGroup**(*images*, *id=None*, *sky=0.0*)
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.

append(*value*)
Appends a *SkyImage* to the group.

calc_sky(*overlap=None*, *delta=True*)
Compute sky background value.

Parameters

- **overlap** (SkyImage, SkyGroup, *SphericalPolygon*, *list of tuples*, *None* (https://docs.python.org/3/library/constants.html#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.

id
Set or get *SkyImage*'s *id*.

While *id* can be of any type, it is prefereble that *id* be of a type with nice string representation.

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 **polygon** – A SphericalPolygon that is the intersection of this *SkyImage* and skyimage.

Return type SphericalPolygon

polygon
Get image's bounding polygon.

radec
Get RA and DEC of the verteces of the bounding polygon as a ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) of shape (N, 2) where N is the number of verteces + 1.

sky
Sky background value. See *calc_sky* for more details.

skystatistics

The skystatistics module contains various statistical functions used by skymatch.

skystatistics module provides statistics computation class used by *match()* and *SkyImage*.

Authors Mihai Cara (contact: help@stsci.edu)

class jwst.skymatch.skystatistics.**SkyStats**(*skystat='mean'*, *lower=None*, *upper=None*, *nclip=5*, *lsig=4.0*, *usig=4.0*, *binwidth=0.1*, ***kwargs*)

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 `skystat`='median' will compute the median of the distribution.
- **lower** (*float* (https://docs.python.org/3/library/functions.html#float), *None* (https://docs.python.org/3/library/constants.html#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* (https://docs.python.org/3/library/constants.html#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`.

calc_sky(*data*)

Computes statistics on data.

Parameters **data** (*numpy.ndarray* (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#nu – 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 comuting the statistics reported in `skyvalue`.

Return type tuple (https://docs.python.org/3/library/stdtypes.html#tuple)

region

The `region` module provides a polygon filling algorithm used by `skymatch` to create data masks.

Polygon filling algorithm.

Authors Nadezhda Dencheva, Mihai Cara (contact: help@stsci.edu)

class jwst.skymatch.region.**Region**(*rid*, *coordinate_system*)
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

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

class jwst.skymatch.region.**Edge**(*name=None*, *start=None*, *stop=None*, *next=None*)
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]

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*)

next

start

stop

ymax

ymin

class jwst.skymatch.region.**Polygon**(*rid*, *vertices*, *coord_system='Cartesian'*)
Represents a 2D polygon region with multiple vertices

Parameters

- **rid** (*string*) – polygon id
- **vertices** (*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

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** –
- **Set the Global Edge Table (GET)** (–) –
- **Set y to be the smallest y coordinate that has an entry in GET** (–) –
- **Initialize the Active Edge Table (AET) to be empty** (–) –
- **For each scan line** (–) –

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.

jwst.skymatch Package

This package provides support for sky background subtraction and equalization (matching).

Classes

SkyMatchStep([name, parent, config_file, ...])	SkyMatchStep: Subtraction or equalization of sky background in science images.

SkyMatchStep

class jwst.skymatch.**SkyMatchStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)
Bases: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

reference_file_types = []

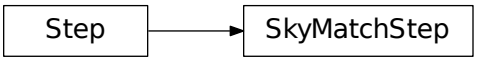
spec = '\n # General sky matching parameters:\n skymethod = option(\'local\', \'global

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



12.1.46 Source Catalog

Description

This step creates a final catalog of source photometry and morphologies.

Source Detection

Sources are detected using image segmentation (http://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 (http://photutils.readthedocs.org/en/latest/photutils/detection.html#source-extraction-using-image-segmentation) and is called the threshold method, where 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 (sigma) 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

Note that overlapping sources are detected as single sources. Separating those sources requires a deblending procedure, such as a multi-thresholding technique used by SExtractor (http://www.astromatic.net/software/sextractor). Here we use the Photutils deblender, which is an experimental 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. Here we use the functions in Photutils (http://photutils.readthedocs.org/en/latest/photutils/segmentation.html). Please see the Photutils SourceProperties (http://photutils.readthedocs.org/en/latest/api/photutils.segmentation.SourceProperties.html#photutils.segmentation.SourceProperties) class for the list of the properties that are calculated for each source.

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: *jwst.stpipe.Step*

Create a final catalog of source photometry and morphologies.

Parameters **input** (str or DrizProductModel) – A FITS filename or a DrizProductModel

of a single drizzled image. The input image is assumed to be background subtracted.

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.
- ****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

spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

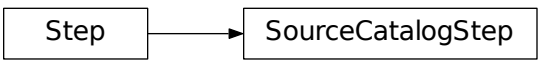
spec = "\n kernel_fwhm = float(default=2.0) # Gaussian kernel FWHM in pixels\n kernel_

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



12.1.47 Source Type (SRCTYPE) Determination

Description

The Source Type (`srctype`) step in the calibration pipeline checks or sets whether a spectroscopic source should be treated as a point or extended object. This information is then used in some subsequent spectroscopic processing steps.

Depending on the JWST observing mode, the observer may have the option of designating a source type in the APT template for the observations. 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 in the APT is passed along to DMS processing, which sets the value of the `SRCTYPE` keyword in the primary header of the level-1b (_uncal.fits) product that's used as input to the calibration pipeline. The `SRCTYPE` keyword may have values of `POINT`, `EXTENDED`, or `UNKNOWN`.

The `srctype` calibration step checks to see if the `SRCTYPE` keyword has been populated and its value. If the observer did not provide a source type value or the `SRCTYPE` keyword is set to `UNKNOWN`, the `srctype` step will choose a suitable default value based on the observing mode of the exposure.

The default values set by the step, as a function of exposure type (the value of the `EXP_TYPE` keyword) is shown in the table below.

EXP_TYPE	Exposure Type	Default 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	Point

For NIRSpec MOS exposures (EXP_TYPE="NRS_MSASPEC"), there are multiple sources per exposure and hence a single 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 is less than zero, the source type is set to `UNKNOWN`. 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 as stored in a `SRCTYPE` keyword located in the header of the SCI extension associated with each source/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.

Step Arguments

The Source Type step has no step-specific arguments.

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: *jwst.stpipe.Step*

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 SRCTYPE keyword value, which is populated from user info in the APT, or the NIRSpec MSA planning tool.

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.
- ****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

spec	

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

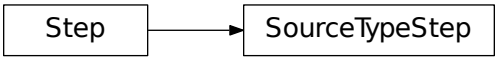
spec = '\n '

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



12.1.48 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 configuration file
- Instantiating the Step directly from Python

Running a Step from a configuration file

A Step configuration file is in the well-known ini-file format. stpipe uses the ConfigObj (https://configobj.readthedocs.io/en/latest/) library to parse them.

Every step configuration file must contain the `name` and `class` of the step, followed by parameters that are specific to the step being run.

`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.

`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`.

Below `name` and `class` in the configuration file are parameters specific to the Step. The set of accepted parameters is defined in the Step's spec member. You can print out a Step's configspec using the `stspec` commandline utility. For example, to print the configspec for an imaginary step called `stpipe.cleanup`:

```
$ stspec stpipe.cleanup
# The threshold below which to apply cleanup
threshold = float()
```

(continues on next page)

(continued from previous page)

```
# A scale factor
scale = float()

# The output file to save to
output_file = output_file(default = None)
```

Note: Configspec information can also be displayed from Python, just call `print_configspec` on any Step class:

```
>>> from jwst.stpipe import cleanup
>>> cleanup.print_configspec()
# The threshold below which to apply cleanup
threshold = float()

# A scale factor
scale = float()
```

Using this information, one can write a configuration file to use this step. For example, here is a configuration file (`do_cleanup.cfg`) that runs the `stpipe.cleanup` step to clean up an image.

```
name = "MyCleanup"
class = "stpipe.cleanup"

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 configuration file
- A Python class

Additional configuration parameters may be passed on the commandline. These parameters override any that are present in the configuration 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 configuration file from above, but override it so the threshold parameter is different:

```
$ strun do_cleanup.cfg 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 configuration file:

```
$ strun -h do_cleanup.cfg
usage: strun [--logcfg LOGCFG] cfg_file_or_class [-h] [--pre_hooks]
             [--post_hooks] [--skip] [--scale] [--extname]

optional arguments:
  -h, --help       show this help message and exit
  --logcfg LOGCFG  The logging configuration file to load
```

(continues on next page)

(continued from previous page)

```
  --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`.

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.

Running a Step in Python

Running a step can also be done inside the Python interpreter and is as simple as calling its `run()` or `call()` classmethods.

run()

The `run()` classmethod will run a previously instantiated step class. This is very useful if one wants to setup the step's attributes first, then run it:

```
from jwst.flatfield import FlatFieldStep

mystep = FlatFieldStep()
mystep.override_sflat = 'sflat.fits'
output = mystep.run(input)
```

Using the *`run()`* method is the same as calling the instance or class directly. They are equivalent:

```
output = mystep(input)
```

call()

If one has all the configuration in a configuration file or can pass the arguments directly to the step, one can use call(), which creates a new instance of the class every time you use the `call()` method. So:

```
output = mystep.call(input)
```

makes a new instance of `FlatFieldStep` and then runs. Because it is a new instance, it ignores any attributes of `mystep` that one may have set earlier, such overriding the sflat.

The nice thing about call() is that it can take a configuration file, so:

```
output = mystep.call(input, config_file='my_flatfield.cfg')
```

and it will take all the configuration from the config file.

Configuration parameters may be passed to the step by setting the `config_file` kwarg in `call` (which takes a path to a configuration file) or as keyword arguments. Any remaining positional arguments are passed along to the step's `process()` method:

```
from jwst.stpipe import cleanup

cleanup.call('image.fits', config_file='do_cleanup.cfg', threshold=42.0)
```

So use call() if you're passing a config file or passing along args or kwargs. Otherwise use run().

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 configuration file or directly from Python.

From a configuration file

A Pipeline configuration file follows the same format as a Step configuration file: the ini-file format used by the ConfigObj (https://configobj.readthedocs.io/en/latest/) library.

Here is an example pipeline configuration file for a `TestPipeline` class:

```
name = "TestPipeline"
class = "stpipe.test.test_pipeline.TestPipeline"

science_filename = "science.fits"
flat_filename = "flat.fits"
output_filename = "output.fits"

[steps]
  [[flat_field]]
    config_file = "flat_field.cfg"
    threshold = 42.0

  [[combine]]
    skip = True
```

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. To figure out what configuration parameters are available, use the `stspec` script (just as with a regular step):

```
> stspec stpipe.test.test_pipeline.TestPipeline
start_step = string(default=None)# Start the pipeline at this step
end_step = string(default=None)# End the pipeline right before this step
science_filename = input_file()  # The input science filename
flat_filename = input_file()     # The input flat filename
skip = bool(default=False)   # Skip this step
output_filename = output_file()  # The output filename
[steps]
[[combine]]
config_file = string(default=None)
skip = bool(default=False)   # Skip this step
[[flat_field]]
threshold = float(default=0.0)# The threshold below which to remove
multiplier = float(default=1.0)# Multiply by this number
skip = bool(default=False)   # Skip this step
config_file = string(default=None)
```

Note that there are some additional optional configuration keys (`start_step` and `end_step`) for controlling when the pipeline starts and stops. This is covered in the section *Running partial Pipelines*.

For each Step's section, the parameters for that step may either be specified inline, or specified by referencing an external configuration file just for that step. For example, a pipeline configuration file that contains:

```
[steps]
  [[flat_field]]
    threshold = 42.0
    multiplier = 2.0
```

is equivalent to:

```
[steps]
  [[flat_field]]
    config_file = myflatfield.cfg
```

with the file `myflatfield.cfg` in the same directory:

```
threshold = 42.0
multiplier = 2.0
```

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 configuration file above:

```
from stpipe.test.test_pipeline import TestPipeline

steps = {
    'flat_field':   {'threshold': 42.0}
    }
```

(continues on next page)

(continued from previous page)

```
pipe = TestPipeline(
    "TestPipeline",
    config_file=__file__,
    science_filename="science.fits",
    flat_filename="flat.fits",
    output_filename="output.fits",
    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 step parameters overridden from the commandline use dot notation to specify the parameter name. For example, to override the `threshold` value on the `flat_field` step in the example pipeline above, one can do:

```
> strun stpipe.test.test_pipeline.TestPipeline --steps.flat_field.threshold=48
```

From Python

Once the pipeline has been configured (as above), just call the instance to run it.

```
pipe()
```

Running partial Pipelines

There are two kinds of pipelines available:

1) Flexible pipelines are written in Python and may contain looping, conditionals and steps with more than one input or output.

2) Linear pipelines have a strict linear progression of steps and only have one input and output.

Linear pipelines have a feature that allows only a part of the pipeline to be run. This is done through two additional configuration parameters: `start_step` and `end_step`. `start_step` specifies the first step to run. `end_step` specifies the last step to run. Like all other configuration parameters, they may be either specified in the Pipeline configuration file, or overridden at the commandline.

When `start_step` and `end_step` indicate that only part of the pipeline will be run, the results of each step will be cached in the current working directory. This allows the pipeline to pick up where it left off later.

Note: In the present implementation, all this caching happens in the current working directory – we probably want a more sane way to manage these files going forward.

Each step may also be skipped by setting its configuration parameter `skip` to True (either in the configuration file or at the command line).

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 configuration 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` configuration parameter 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]
  [[flat_field]]
    threshold = 42.0
    post_hooks = "ds9 {0}",
```

Logging

Log messages are emmitted from each Step at different levels of importance. The levels used are the standard ones for Python (from least important to most important:

- DEBUG
- INFO
- WARNING
- ERROR
- 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:

- The file specified with the `--logcfg` option to the `strun` script.
- A file called `stpipe-log.cfg` in the current working directory.
- `~/stpipe-log.cfg`
- `/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:

- `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.
- `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`.
- `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.
- `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
```

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 configuration 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.

Configuration 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 configuration parameter.

The configuration parameters are defined by the *Step.spec* member.

Input Files, Associations, and Directories

It is presumed that all input files are co-resident 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 configuration 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. Explicity by specifying the parameter `save_results`
3. Explicity by specifying a file name with the paremeter `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 the default coniguration file, or within the code itself. See `calwebb_dark` for an example of specifying in the configuration.

For an example where the suffix can only be determined at runtime, see `calwebb_sloper`. 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 configuration parameters.

1. Objects from other Steps in a pipeline are passed as arguments to the `process` method.
2. The configuration 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 configuration 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 configuration 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 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")
```

(continues on next page)

(continued from previous page)

```
        # 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 configuration parameters. It is in the `configspec` format defined in the `ConfigObj` library that stpipe uses.

The `configspec` format defines the types of the configuration parameters, as well as allowing an optional tree structure.

The types of configuration 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.

Configuration 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 configuration 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 configuration 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 configuration parameters to the Step's configspec.

For each reference file type that the Step declares, an `override_*` configuration 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 configuration 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

# 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

```

(continues on next page)

(continued from previous page)

```
> ExampleStep --config-file=example_step.cfg

> ExampleStep --parameter1=42.0 input_file.fits
```

Pipelines

Writing a Pipeline

There are two ways to go about writing a pipeline depending on how much flexibility is required.

1. A linear pipeline defines a simple linear progression of steps where each step has a single input and a single output flowing directly into the next step.

2. A flexible pipeline allows the pipeline to be defined in Python code and all of the tools that implies, such as loops, conditionals and multiple inputs and/or outputs.

Linear pipeline

To create a linear pipeline, one inherits from the `LinearPipeline` class and adds a special member `pipeline_steps` to define the order of the steps:

```
from jwst.stpipe import LinearPipeline

# Some locally-defined steps
from . import FlatField, RampFitting

class ExampleLinearPipeline(LinearPipeline):
    """
    This example linear pipeline has only two steps.
    """
    pipeline_steps = [
        ('flat_field', FlatField),
        ('ramp_fitting', RampFitting)
        ]
```

The `pipeline_steps` member is a list of tuples. Each tuple is a pair (*name*, *class*) where *name* is the name of the specific step, and *class* is the step's class. Both are required so the same step class can be used multiple times in the pipeline. The name is also used for the section headings in the pipeline's configuration file.

Flexible pipeline

The basics of writing a flexible 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 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 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,
        '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 configuration 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()

# 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.

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:

```
>>> 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`
```

(continues on next page)

(continued from previous page)

```
    # 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 the configuration 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 *`DataModel`* 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.

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

In general, all input, 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 configuration 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 the `cfg` file or as a command-line option.
- Implicitly when a step is called by `strun`.
- Implicitly when the configuration option `output_file` is given a value.

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` configuration parameter.
- Explicitly in the code. Often this is done in 'Pipeline's 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` configuration 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` configuration option is given.

Sub-Steps and Output

Normally, the value of a configuration option is completely local to the `Step`: A `Step`, called from another `Step` or `Pipeline`, can only access its own configuration 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 `outlier_detection` step. For such steps, one can prevent using the `Pipeline.output_file` specification by setting the configuration 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. This can be seen in the `calwebb_spec3` default configuration file:

```
name = "Spec3Pipeline"
class = "jwst.pipeline.Spec3Pipeline"

    [steps]
      [[mrs_imatch]]
```

(continues on next page)

(continued from previous page)

```
        suffix = 'mrs_imatch'
      [[outlier_detection]]
        suffix = 'crf'
        save_results = True
        search_output_file = False
      [[resample_spec]]
        suffix = 's2d'
        save_results = True
      [[cube_build]]
        suffix = 's3d'
        save_results = True
      [[extract_1d]]
        suffix = 'x1d'
        save_results = True
```

Output API: When More Control Is Needed

In summary, the standard output API, as described so far, is basically "set a few configuration 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.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([name, parent, config_file, _validate_kwds])	Create a *Step* instance.
Pipeline(*args, **kwargs)	A Pipeline is a way of combining a number of steps together.
LinearPipeline(*args, **kwargs)	A LinearPipeline is a way of combining a number of steps together in a simple linear order.

Step

class jwst.stpipe.**Step**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `object` (https://docs.python.org/3/library/functions.html#object)

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.

- ****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

input_dir	
make_output_path	Return function that creates the output path
prefetch_references	
reference_file_types	
spec	

Methods Summary

__call__(*args)	Run handles the generic setup and teardown that happens with the running of each step.
call(*args, **kwargs)	Creates and runs a new instance of the class.
closeout([to_close, to_del])	Close out step processing
default_output_file([input_file])	Create a default filename based on the input name
default_suffix()	Return a default suffix based on the step
from_cmdline(args)	Create a step from a configuration file.
from_config_file(config_file[, parent, name])	Create a step from a configuration file.
from_config_section(config[, parent, name, ...])	Create a step from a configuration file fragment.
get_ref_override(reference_file_type)	Determine and return any override for `reference_file_type`.
get_reference_file(input_file, ...)	Get a reference file from CRDS.
load_as_level2_asn(obj)	Load object as an association
load_as_level3_asn(obj)	Load object as an association
load_spec_file([preserve_comments])	
make_input_path(file_path)	Create an input path for a given file path
merge_config(config, config_file)	
open_model(obj)	Open a datamodel
print_configspec([stream])	
process(*args)	This is where real work happens.
reference_uri_to_cache_path(reference_uri)	Convert an abstract CRDS reference URI to an absolute file path in the CRDS cache.
resolve_file_name(file_name)	Resolve a file name expressed relative to this Step's configuration file.
run(*args)	Run handles the generic setup and teardown that happens with the running of each step.
save_model(model[, suffix, idx, ...])	Saves the given model using the step/pipeline's naming scheme
search_attr(attribute[, default, parent_first])	Return first non-None attribute in step heirarchy
set_primary_input(obj[, exclusive])	Sets the name of the master input file and input directory.

Attributes Documentation

input_dir

make_output_path

Return function that creates the output path

prefetch_references = True

reference_file_types = []

spec = "\n pre_hooks = string_list(default=list())\n post_hooks = string_list(default=

Methods Documentation

__call__(**args*)
Run handles the generic setup and teardown that happens with the running of each step. The real work that is unique to each step type is done in the *process* method.

classmethod call(**args*, ***kwargs*)
Creates and runs a new instance of the class.

To set configuration parameters, pass a `config_file` path or keyword arguments. Keyword arguments override those in the specified `config_file`.

Any positional `*args` will be passed along to the step's *process* method.

Note: this method creates a new instance of *Step* with the given `config_file` if supplied, plus any extra `*args` and `**kwargs`. If you create an instance of a Step, set parameters, and then use this *call()* method, it will ignore previously-set parameters, as it creates a new instance of the class with only the `config_file`, `*args` and `**kwargs` passed to the *call()* method.

If not used with a `config_file` or specific `*args` and `**kwargs`, it would be better to use the *run* method, which does not create a new instance but simply runs the existing instance of the *Step* class.

closeout(*to_close=None*, *to_del=None*)
Close out step processing

Parameters

- **to_close** (*[object* (https://docs.python.org/3/library/functions.html#object) *(, ..)]*) – List of objects with a `close` method to execute The objects will also be deleted
- **to_del** (*[object* (https://docs.python.org/3/library/functions.html#object) *(, ..)]*) – List of objects to simply delete

Notes

Other operations, such as forced garbage collection will also be done.

default_output_file(*input_file=None*)
Create a default filename based on the input name

default_suffix()
Return a default suffix based on the step

static from_cmdline(*args*)
Create a step from a configuration file.

Parameters **args** (*list of str*) – Commandline arguments

Returns

step – If the config file has a `class` parameter, the return value will be as instance of that class.

Any parameters found in the config file will be set as member variables on the returned *Step* instance.

Return type Step instance

classmethod from_config_file(*config_file*, *parent=None*, *name=None*)
Create a step from a configuration file.

Parameters

- **config_file** (*path or readable file-like object*) – The config file to load parameters from
- **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.
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – If provided, use that name for the returned instance. If not provided, the following are tried (in order): - The `name` parameter in the config file - The filename of the config file - The name of returned class

Returns

step – If the config file has a `class` parameter, the return value will be as instance of that class. The `class` parameter in the config file must specify a subclass of `cls`. If the configuration file has no `class` parameter, then an instance of `cls` is returned.

Any parameters found in the config file will be set as member variables on the returned *Step* instance.

Return type Step instance

classmethod from_config_section(*config*, *parent=None*, *name=None*, *config_file=None*)
Create a step from a configuration file fragment.

Parameters

- **config** (*configobj.Section instance*) – The config file fragment containing parameters for this step only.
- **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.
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – If provided, use that name for the returned instance. If not provided, try the following (in order): - The `name` parameter in the config file fragment - The name of returned class
- **config_file** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *optional*) – The path to the config file that created this step, if any. This is used to resolve relative file name parameters in the config file.

Returns **step** – Any parameters found in the config file fragment will be set as member variables on the returned *Step* instance.

Return type instance of cls

get_ref_override(*reference_file_type*)
Determine and return any override for `reference_file_type`.

Returns

Return type override_filepath or None.

get_reference_file(*input_file*, *reference_file_type*)
Get a reference file from CRDS.

If the configuration file or commandline parameters override the reference file, it will be automatically used when calling this function.

Parameters

- **input_file** (*jwst.datamodels.ModelBase instance*) – A model of the input file. Metadata on this input file will be used by the CRDS "bestref" algorithm to obtain a reference file.
- **reference_file_type** (*string*) – The type of reference file to retrieve. For example, to retrieve a flat field reference file, this would be 'flat'.

Returns **reference_file**

Return type path of reference file, a string

load_as_level2_asn(*obj*)
Load object as an association

Loads the specified object into a Level2 association. If necessary, prepend *Step.input_dir* to all members.

Parameters **obj** (*object* (https://docs.python.org/3/library/functions.html#object)) – Object to load as a Level2 association

Returns **association** – Association

Return type jwst.associations.lib.rules_level2_base.DMSLevel2bBase

load_as_level3_asn(*obj*)
Load object as an association

Loads the specified object into a Level3 association. If necessary, prepend *Step.input_dir* to all members.

Parameters **obj** (*object* (https://docs.python.org/3/library/functions.html#object)) – Object to load as a Level3 association

Returns **association** – Association

Return type jwst.associations.lib.rules_level3_base.DMS_Level3_Base

classmethod load_spec_file(*preserve_comments=False*)

make_input_path(*file_path*)
Create an input path for a given file path

If `file_path` has no directory path, use `self.input_dir` as the directory path.

Parameters **file_path** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or obj*) – The supplied file path to check and modify. If anything other than `str` (https://docs.python.org/3/library/stdtypes.html#str), the object is simply passed back.

Returns **full_path** – File path using *input_dir* if the input had no directory path.

Return type str (https://docs.python.org/3/library/stdtypes.html#str) or obj

classmethod merge_config(*config*, *config_file*)

open_model(*obj*)
Open a datamodel

Primarily a wrapper around `DataModel.open` to handle *Step* peculiarities

Parameters **obj** (*object* (https://docs.python.org/3/library/functions.html#object)) – The object to open

Returns **datamodel** – Object opened as a datamodel

Return type *DataModel*

classmethod print_configspec(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

process(**args*)

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

reference_uri_to_cache_path(*reference_uri*)

Convert an abstract CRDS reference URI to an absolute file path in the CRDS cache. Reference URI's are typically output to dataset headers to record the reference files used.

e.g. 'crds://jwst_miri_flat_0177.fits' –> '/grp/crds/cache/references/jwst/jwst_miri_flat_0177.fits'

The CRDS cache is typically located relative to env var CRDS_PATH with default value /grp/crds/cache. See also https://jwst-crds.stsci.edu

resolve_file_name(*file_name*)

Resolve a file name expressed relative to this Step's configuration file.

run(**args*)

Run handles the generic setup and teardown that happens with the running of each step. The real work that is unique to each step type is done in the *process* method.

save_model(*model*, *suffix=None*, *idx=None*, *output_file=None*, *force=False*, *format=None*, ***components*)

Saves the given model using the step/pipeline's naming scheme

Parameters

- **model** (*jwst.datamodels.Model instance*) – The model to save.
- **suffix** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The suffix to add to the filename.
- **idx** (*object* (https://docs.python.org/3/library/functions.html#object)) – Index identifier.
- **output_file** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – Use this file name instead of what the Step default would be.
- **force** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – Regardless of whether `save_results` is `False` (https://docs.python.org/3/library/constants.html#False) and no `output_file` is specified, try saving.
- **format** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The format of the file name. This is a format string that defines where `suffix` and the other components go in the file name.
- **components** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – Other components to add to the file name.

Returns **output_paths** – List of output file paths the model(s) were saved in.

Return type [str (https://docs.python.org/3/library/stdtypes.html#str)[, ..]]

search_attr(*attribute*, *default=None*, *parent_first=False*)

Return first non-None attribute in step heirarchy

Parameters

- **attribute** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – The attribute to retrieve
- **default** (*obj*) – If attribute is not found, the value to use
- **parent_first** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If `True` (https://docs.python.org/3/library/constants.html#True), allow parent definition to override step version

Returns **value** – Attribute value or `default` if not found

Return type obj

set_primary_input(*obj*, *exclusive=True*)

Sets the name of the master input file and input directory. Used to generate output file names.

Parameters

- **obj** (*str* (https://docs.python.org/3/library/stdtypes.html#str) *or* `DataModel`) – The object to base the name on. If a datamodel, use Datamodel.meta.filename.
- **exclusive** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – If True, only set if an input name is not already used by a parent Step. Otherwise, always set.

Pipeline

class jwst.stpipe.**Pipeline**(**args*, ***kwargs*)

Bases: *jwst.stpipe.Step*

A Pipeline is a way of combining a number of steps together.

See `Step.__init__` for the parameters.

Attributes Summary

spec
step_defs

Methods Summary

get_ref_override(reference_file_type)	Return any override for `reference_file_type` for any of the steps in Pipeline `self`.
load_spec_file([preserve_comments])	
merge_config(config, config_file)	
set_input_filename(path)	

Attributes Documentation

spec = '\n '

step_defs = {}

Methods Documentation

get_ref_override(*reference_file_type*)
Return any override for `reference_file_type` for any of the steps in Pipeline `self`. OVERRIDES Step.

Returns

Return type override_filepath or None.

classmethod load_spec_file(*preserve_comments=False*)

classmethod merge_config(*config*, *config_file*)

set_input_filename(*path*)

LinearPipeline

class jwst.stpipe.**LinearPipeline**(**args*, ***kwargs*)
Bases: *jwst.stpipe.Pipeline*

A LinearPipeline is a way of combining a number of steps together in a simple linear order.

See `Step.__init__` for the parameters.

Attributes Summary

pipeline_steps
spec
step_defs

Methods Summary

process(input_file)	Run the pipeline.
set_input_filename(path)	

Attributes Documentation

pipeline_steps = None

spec = '\n # start_step and end_step allow only a part of the pipeline to run\n start_

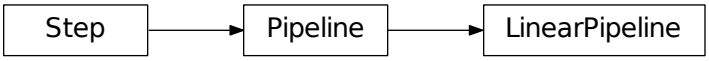
step_defs = {}

Methods Documentation

process(*input_file*)
Run the pipeline.

set_input_filename(*path*)

Class Inheritance Diagram



12.1.49 Stray Light Correction

Description

Assumption

The current stray-light correction is only valid for MIRI MRS Short wavelength data. The straylight step uses information about which pixels belong to a slice and which pixels are located in the slice gaps. This informations is contained meta data of the input image and was loaded from a reference file by the assign_wcs step. Thus running the assign_wcs on the input data is a prerequisite to the straylight step.

Overview

This routine removes and/or otherwise corrects for stray-light that may contaminate a MIRI MRS short-wavelength spectrum, due a bright source in the MRS slice gaps. The current routine determines the stray-light by using signal in-between slices and interpolates over the slice.

The chief source of the MIRI MRS stray-light appears to be caused by scattering in optical components within the SMO. The stray-light is manifested as a signal that extends in the detector row direction. Its magnitude is proportional to that of bright illuminated regions of the spectral image, at a ratio that falls with increasing wavelength, from about 1 % in Channel 1A to undetectable low levels long-ward of Channel 2B. Two components of the stray-light have been observed, a smooth and a structured distribution.

Algorithm

The basic idea of the stray-light removal algorithm is to only deal with the smooth component of the stray-light. Due to the extended nature of the stray-light we use the detected signal in the slice gaps, where nominally no photons should hit the detectors, and assume that all detected light is the stray-light. Using this measurement, we can interpolate the gap flux within the slice to estimate the amount of the stray-light in the slice.

There are two possible algorithms in the stray-light step. The first algorithm is a more simplistic approach by dealing with the stray-light estimation row-by-row and interpolating the gap flux linearly. An intermediate stray-light map is generated row-by-row and then this map is further smoothed to remove row-by-row variations. This algorithm uses a stray-light mask reference file that contains 1s for gap pixels and 0s for science pixels.

Given the extended nature of the smooth component of the MRS stray-light, it is obvious that a row-by-row handling of the stray-light could be replaced by a two-dimensional approach such that no additional smoothing is required. For the second algorithm we improved the technique by using the Modified Shepard's Method to interpolate the gap fluxes two dimensionally. The stray-light correction for each science pixel is based on the flux of the gap pixels with a "region of influence" from the science pixel. The algorithm takes each science pixel and determines the amount of

stray-light to remove from the pixel s by interpolating the fluxes p_i measured by the gap pixels. The gap pixel flux is weighted by the distance d_i between the science pixel and gap pixel. The Modified Shepard's Method uses this distance to weight the different contributors according the equation:

$$s = \frac{\sum_{i=1}^{n} p_i w_i}{\sum_{i=1}^{n} w_i}$$

$$w_i = \frac{max(0, R - d_i)^k}{R d_i}$$

The radius of influence R and the exponent k are variables that can be adjusted to the actual problem. The default values for these parameters are $R = 50$ pixels and $k = 1$.

Reference File Types

The default algorithm in the MIRI MRS stray-light correction step uses information contained in the meta data of the input image which maps each pixels to a slice or the region between the slices, also known as the slice gaps. This information was previously loaded from a reference file into the meta data by the assign_wcs step. There is an option to use a more simplistic algorithm that uses stray-light mask reference file.

CRDS Selection Criteria

If –method = "Nearest" option is used then the MIRI MRS stray-light reference file is selected on the basis of INSTRUME, DETECTOR, and BAND values of the input science data set.

MIRI MRS stray-light Reference File Format

The stray-light mask reference files are FITS files with and empty primary data array and one IMAGE extension. This IMAGE extension is a 2-D integer image mask file of size 1032 X 1024. The mask contains values of 1 for pixels that fall in the slice gaps and values of 0 for science pixels. The stray-light algorithm only uses pixels that fall in the slice gaps to determine the correction.

Step Arguments

There are two possible algorithms to use for the stray-light correction step. The first one is more simplistic and uses a row-row interpolation of the gap pixels to determine the stray-light correction. The second algorithm uses a 2-D approach by using a Modified Shepard's Method to interpolate the light in the gap pixels. The default algorithm is to use the second method. The first method was kept for comparison to the second method and may be removed in a future version.

The argument which sets which algorithm to use is

- `--method [string]`

The default Modified Shepard's Method has the value 'ModShepard'. To set the step to use the simplistic row-row interpolate use 'Nearest'.

There are two arguments if the Modified Shepard's Method is being used. These are

- `--roi [integer]`

This parameter sets the 'radius of influence' to select the gap pixels to be used in the correction. The default value is set to 50 pixels.

- `--power [integer]`

This parameter is the power k in the Modified Shepard's Method weighting equation. The default value is set to 1.

The usage of both parameters are shown in the description of the Modified Shepard's Method distance weighting equation:

$$s = \frac{\sum_{i=1}^{n} p_i w_i}{\sum_{i=1}^{n} w_i}$$

where,

$$w_i = \frac{max(0, R - d_i)^k}{R d_i}$$

The radius of influence R and the exponent k are variables that can be adjusted to the actual problem. The default values for these parameters are $R = 50$ pixels and $k = 1$.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

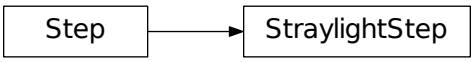
reference_file_types = ['straymask']

spec = "\n method = option('Nearest','ModShepard',default='ModShepard') #Algorithm meth

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



12.1.50 Superbias Subtraction

Description

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 bit-wise OR operation.

The ERR arrays in the science ramp data are unchanged.

Subarrays

If the subarray mode of the superbias reference file matches that of the science exposure, the reference data are applied as-is. If the superbias reference file contains full-frame data, while the science exposure is a subarray mode, the corresponding subarray will be extracted from the superbias reference data before being applied.

Reference File Types

The superbias subtraction step uses a SUPERBIAS reference file.

CRDS Selection Criteria

Superbias reference files are selected on the basis of the INSTRUME, DETECTOR, READPATT and SUBARRAY values of the input science data set.

SUPERBIAS Reference File Format

Superbias 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	2	ncols x nrows	float
ERR	2	ncols x nrows	float
DQ	2	ncols x nrows	integer

The BINTABLE extension contains the bit assignments used in the DQ array. It uses `EXTNAME=DQ_DEF` and contains 4 columns:

- BIT: integer value giving the bit number, starting at zero
- VALUE: the equivalent base-10 integer value of BIT
- NAME: the string mnemonic name of the data quality condition
- DESCRIPTION: a string description of the condition

Step Arguments

The superbias subtraction step has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

reference_file_types = ['superbias']

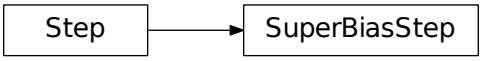
spec = '\n\n '

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



12.1.51 Transforms

TPCorr

jwst.transforms.tpcorr Module

A module that provides *TPCorr* class - a `Model` derived class that applies linear tangent-plane corrections to V2V3 coordinates of JWST instrument's WCS.

Authors Mihai Cara (contact: help@stsci.edu)

Functions

rot_mat3D(angle, axis)	

rot_mat3D

`jwst.transforms.tpcorr.`**`rot_mat3D`**(*angle*, *axis*)

Classes

IncompatibleCorrections	An exception class used to report cases when two or more tangent plane corrections cannot be combined into a single one.
TPCorr([v2ref, v3ref, roll, matrix, shift])	Apply `V2ref`, `V3ref`, and `roll` to input angles and project the point from the tangent plane onto a celestial sphere.

IncompatibleCorrections

exception `jwst.transforms.tpcorr.`**`IncompatibleCorrections`**

An exception class used to report cases when two or more tangent plane corrections cannot be combined into a single one.

TPCorr

class jwst.transforms.tpcorr.**TPCorr**(*v2ref=0.0, v3ref=0.0, roll=0.0, matrix=[[1.0, 0.0], [0.0, 1.0]], shift=[0.0, 0.0], **kwargs*)

Bases: astropy.modeling.core.Model

Apply V2ref, V3ref, and roll to input angles and project the point from the tangent plane onto a celestial sphere.

Parameters

- **v2ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – V2 position of the reference point in degrees. Default is 0 degrees.
- **v3ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – V3 position of the reference point in degrees. Default is 0 degrees.
- **roll** (*float* (https://docs.python.org/3/library/functions.html#float)) – Roll angle in degrees. Default is 0 degrees.

Attributes Summary

input_units	This property is used to indicate what units or sets of units the evaluate method expects, and returns a dictionary mapping inputs to units (or None (https://docs.python.org/3/library/constants.html#None) if any units are accepted).
inputs	
matrix	
outputs	
param_names	
r0	Radius of the generating sphere.
return_units	This property is used to indicate what units or sets of units the output of evaluate should be in, and returns a dictionary mapping outputs to units (or None (https://docs.python.org/3/library/constants.html#None) if any units are accepted).
roll	
shift	
standard_broadcasting	
v2ref	
v3ref	

Methods Summary

__call__(v2, v3[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
cartesian2spherical(x, y, z)	Convert cartesian coordinates to spherical coordinates (in acrsec).
combine(t2, t1)	Combine transformation t2 with another transformation (t1) *previously applied* to the coordinates.

Continued on next page

Table 345 – continued from previous page

evaluate(v2, v3, v2ref, v3ref, roll, matrix, ...)	Evaluate the model on some input variables.
spherical2cartesian(alpha, delta)	Convert spherical coordinates (in arcsec) to cartesian.
tanp_to_v2v3(xt, yt)	Converts tangent plane coordinates to V2V3 spherical coordinates.
v2v3_to_tanp(v2, v3)	Converts V2V3 spherical coordinates to tangent plane coordinates.

Attributes Documentation

input_units
This property is used to indicate what units or sets of units the evaluate method expects, and returns a dictionary mapping inputs to units (or None (https://docs.python.org/3/library/constants.html#None) if any units are accepted).

Model sub-classes can also use function annotations in evaluate to indicate valid input units, in which case this property should not be overriden since it will return the input units based on the annotations.

inputs = ('v2', 'v3')

matrix

outputs = ('v2c', 'v3c')

param_names = ('v2ref', 'v3ref', 'roll', 'matrix', 'shift')

r0 = 206264.80624709636
Radius of the generating sphere. This sets the circumference to 360 deg so that arc length is measured in deg.

return_units
This property is used to indicate what units or sets of units the output of evaluate should be in, and returns a dictionary mapping outputs to units (or None (https://docs.python.org/3/library/constants.html#None) if any units are accepted).

Model sub-classes can also use function annotations in evaluate to indicate valid output units, in which case this property should not be overriden since it will return the return units based on the annotations.

roll

shift

standard_broadcasting = False

v2ref

v3ref

Methods Documentation

__call__(*v2*, *v3*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static cartesian2spherical(*x*, *y*, *z*)
Convert cartesian coordinates to spherical coordinates (in acrsec).

classmethod combine(*t2*, *t1*)
Combine transformation `t2` with another transformation (`t1`) *previously applied* to the coordinates. That is, transformation `t2` is assumed to *follow* (=applied after) the transformation provided by the argument `t1`.

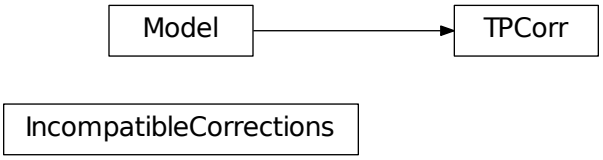
evaluate(*v2*, *v3*, *v2ref*, *v3ref*, *roll*, *matrix*, *shift*)
Evaluate the model on some input variables.

static spherical2cartesian(*alpha*, *delta*)
Convert spherical coordinates (in arcsec) to cartesian.

tanp_to_v2v3(*xt*, *yt*)
Converts tangent plane coordinates to V2V3 spherical coordinates.

v2v3_to_tanp(*v2*, *v3*)
Converts V2V3 spherical coordinates to tangent plane coordinates.

Class Inheritance Diagram



jwst.transforms Package

Functions

rot_mat3D(angle, axis)	

rot_mat3D

jwst.transforms.**rot_mat3D**(*angle*, *axis*)

Classes

AngleFromGratingEquation(groove_density, ...)	Solve the 3D Grating Dispersion Law for the refracted angle.
WavelengthFromGratingEquation(...)	Solve the 3D Grating Dispersion Law for the wavelength.
Unitless2DirCos(*args[, meta, name])	Transform a vector to directional cosines.
DirCos2Unitless(*args[, meta, name])	Transform directional cosines to vector.

Continued on next page

Table 347 – continued from previous page

Rotation3DToGWA(angles, axes_order[, name])	Perform a 3D rotation given an angle in degrees.
Gwa2Slit(slits, models)	NIRSpec GWA to slit transform.
Slit2Msa(slits, models)	NIRSpec slit to MSA transform.
Snell(angle, kcoef, lcoef, tcoef, tref, ...)	Apply transforms, including Snell law, through the NIRSpec prism.
Logical(condition, compareto, value, **kwargs)	Substitute values in an array where the condition is evaluated to True.
NirissSOSSModel(spectral_orders, models)	NIRISS SOSS wavelength solution implemented as a Model.
V23ToSky(angles, axes_order[, name])	Transform from V2V3 to a standard coordinate system (ICRS).
Slit(name, shutter_id, xcen, ycen, ymin, ...)	Nirspec Slit structure definition
NIRCAMForwardRowGrismDispersion(orders[, ...])	Return the transform from grism to image for the given spectral order.
NIRCAMForwardColumnGrismDispersion(orders[, ...])	Return the transform from grism to image for the given spectral order.
NIRCAMBackwardGrismDispersion(orders[, ...])	Return the valid pixel(s) and wavelengths given center x,y and lam
MIRI_AB2Slice([beta_zero, beta_del, channel])	MIRI MRS alpha, beta to slice transform
GrismObject	Grism Objects identified from a direct image catalog and segment map.
NIRISSForwardRowGrismDispersion(orders[, ...])	This model calculates the dispersion extent of NIRISS pixels.
NIRISSForwardColumnGrismDispersion(orders[, ...])	This model calculates the dispersion extent of NIRISS pixels.
NIRISSBackwardGrismDispersion(orders[, ...])	This model calculates the dispersion extent of NIRISS pixels.
V2V3ToIdeal(v3idlyangle, v2ref, v3ref, vparity)	Performs the transform from telescope V2,V3 to Ideal coordinate system.
IdealToV2V3(v3idlyangle, v2ref, v3ref, vparity)	Performs the transform from Ideal to telescope V2,V3 coordinate system.
IncompatibleCorrections	An exception class used to report cases when two or more tangent plane corrections cannot be combined into a single one.
TPCorr([v2ref, v3ref, roll, matrix, shift])	Apply `V2ref`, `V3ref`, and `roll` to input angles and project the point from the tangent plane onto a celestial sphere.

AngleFromGratingEquation

class jwst.transforms.**AngleFromGratingEquation**(*groove_density*, *order*, ***kwargs*)

Bases: astropy.modeling.core.Model

Solve the 3D Grating Dispersion Law for the refracted angle.

Parameters

- **groove_density** (*int* (https://docs.python.org/3/library/functions.html#int)) – Grating ruling density.
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – Spectral order.

Attributes Summary

groove_density	Grating ruling density.
inputs	Wavelength and 3 angle coordinates going into the grating.
order	Spectral order.
outputs	Three angles coming out of the grating.
param_names	

Methods Summary

__call__(lam, alpha_in, beta_in, z[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(lam, alpha_in, beta_in, z, ...)	Evaluate the model on some input variables.

Attributes Documentation

groove_density
Grating ruling density.

inputs = ('lam', 'alpha_in', 'beta_in', 'z')
Wavelength and 3 angle coordinates going into the grating.

order
Spectral order.

outputs = ('alpha_out', 'beta_out', 'zout')
Three angles coming out of the grating.

param_names = ('groove_density', 'order')

Methods Documentation

__call__(*lam*, *alpha_in*, *beta_in*, *z*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*lam*, *alpha_in*, *beta_in*, *z*, *groove_density*, *order*)
Evaluate the model on some input variables.

WavelengthFromGratingEquation

class jwst.transforms.**WavelengthFromGratingEquation**(*groove_density*, *order*, ***kwargs*)

Bases: astropy.modeling.core.Model

Solve the 3D Grating Dispersion Law for the wavelength.

Parameters

- **groove_density** (*int* (https://docs.python.org/3/library/functions.html#int)) – Grating ruling density.

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

Attributes Summary

groove_density	Grating ruling density.
inputs	three angle - alpha_in and beta_in going into the grating and alpha_out coming out of the grating.
order	Spectral order.
outputs	Wavelength.
param_names	

Methods Summary

__call__(alpha_in, beta_in, alpha_out[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(alpha_in, beta_in, alpha_out, ...)	Evaluate the model on some input variables.

Attributes Documentation

groove_density
Grating ruling density.

inputs = ('alpha_in', 'beta_in', 'alpha_out')
three angle - alpha_in and beta_in going into the grating and alpha_out coming out of the grating.

order
Spectral order.

outputs = ('lam',)
Wavelength.

param_names = ('groove_density', 'order')

Methods Documentation

__call__(*alpha_in*, *beta_in*, *alpha_out*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*alpha_in*, *beta_in*, *alpha_out*, *groove_density*, *order*)
Evaluate the model on some input variables.

Unitless2DirCos

class jwst.transforms.**Unitless2DirCos**(**args*, *meta=None*, *name=None*, ***kwargs*)
Bases: astropy.modeling.core.Model

Transform a vector to directional cosines.

Attributes Summary

inputs	
outputs	

Methods Summary

__call__(x, y[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y)	Evaluate the model on some input variables.

Attributes Documentation

inputs = ('x', 'y')

outputs = ('x', 'y', 'z')

Methods Documentation

__call__(*x*, *y*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*)
Evaluate the model on some input variables.

DirCos2Unitless

class jwst.transforms.**DirCos2Unitless**(**args*, *meta=None*, *name=None*, ***kwargs*)
Bases: astropy.modeling.core.Model

Transform directional cosines to vector.

Attributes Summary

inputs	
outputs	

Methods Summary

__call__(x, y, z[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, z)	Evaluate the model on some input variables.

Attributes Documentation

inputs = ('x', 'y', 'z')

outputs = ('x', 'y')

Methods Documentation

__call__(*x*, *y*, *z*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *z*)
Evaluate the model on some input variables.

Rotation3DToGWA

class jwst.transforms.**Rotation3DToGWA**(*angles*, *axes_order*, *name=None*)
Bases: astropy.modeling.core.Model

Perform a 3D rotation given an angle in degrees.

Positive angles represent a counter-clockwise rotation and vice-versa.

Parameters

- **angles** (*array-like*) – Angles of rotation in deg in the order of axes_order.
- **axes_order** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A sequence of 'x', 'y', 'z' corresponding of axis of rotation/

Attributes Summary

angles
inputs
outputs
param_names
separable
standard_broadcasting

Methods Summary

__call__(x, y, z[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, z, angles)	Apply the rotation to a set of 3D Cartesian coordinates.

Attributes Documentation

angles

inputs = ('x', 'y', 'z')

outputs = ('x', 'y', 'z')

param_names = ('angles',)

separable = False

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *z*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *z*, *angles*)
Apply the rotation to a set of 3D Cartesian coordinates.

Gwa2Slit

class jwst.transforms.**Gwa2Slit**(*slits*, *models*)
Bases: astropy.modeling.core.Model

NIRSpec GWA to slit transform.

Parameters

- **slits** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – A list of open slits. A slit is a namedtupe of type *Slit* Slit("name", "shutter_id", "xcen", "ycen", "ymin", "ymax", "quadrant", "source_id", "shutter_state", "source_name", "source_alias", "stellarity", "source_xpos", "source_ypos"])
- **models** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – List of models (Model) corresponding to the list of slits.

Attributes Summary

inputs	Name of the slit and the three angle coordinates at the GWA going from detector to sky.
outputs	Name of the slit, x and y coordinates within the virtual slit and wavelength.
slits	

Methods Summary

__call__(name, angle1, angle2, angle3[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(name, x, y, z)	Evaluate the model on some input variables.
get_model(name)	

Attributes Documentation

inputs = ('name', 'angle1', 'angle2', 'angle3')
Name of the slit and the three angle coordinates at the GWA going from detector to sky.

outputs = ('name', 'x_slit', 'y_slit', 'lam')
Name of the slit, x and y coordinates within the virtual slit and wavelength.

slits

Methods Documentation

__call__(*name*, *angle1*, *angle2*, *angle3*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*name*, *x*, *y*, *z*)
Evaluate the model on some input variables.

get_model(*name*)

Slit2Msa

class jwst.transforms.**Slit2Msa**(*slits*, *models*)
Bases: astropy.modeling.core.Model

NIRSpec slit to MSA transform.

Parameters

- **slits** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – A list of open slits. A slit is a namedtupe, *Slit* Slit("name", "shutter_id", "xcen", "ycen", "ymin", "ymax", "quadrant", "source_id", "shutter_state", "source_name", "source_alias", "stellarity", "source_xpos", "source_ypos")
- **models** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – List of models (Model) corresponding to the list of slits.

Attributes Summary

inputs	Name of the slit, x and y coordinates within the virtual slit.
outputs	x and y coordinates in the MSA frame.
slits	

Methods Summary

__call__(name, x_slit, y_slit[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(name, x, y)	Evaluate the model on some input variables.
get_model(name)	

Attributes Documentation

inputs = ('name', 'x_slit', 'y_slit')
Name of the slit, x and y coordinates within the virtual slit.

outputs = ('x_msa', 'y_msa')
x and y coordinates in the MSA frame.

slits

Methods Documentation

__call__(*name*, *x_slit*, *y_slit*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*name*, *x*, *y*)
Evaluate the model on some input variables.

get_model(*name*)

Snell

class jwst.transforms.**Snell**(*angle*, *kcoef*, *lcoef*, *tcoef*, *tref*, *pref*, *temperature*, *pressure*, *name=None*)
Bases: astropy.modeling.core.Model

Apply transforms, including Snell law, through the NIRSpec prism.

Parameters

- **angle** (*flaot*) – Prism angle in deg.
- **kcoef** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – K coefficients in Sellmeir equation.
- **lcoef** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – L coefficients in Sellmeir equation.
- **tcoef** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – Thermal coefficients of glass.
- **tref** (*float* (https://docs.python.org/3/library/functions.html#float)) – Refernce temperature in K.
- **pref** (*float* (https://docs.python.org/3/library/functions.html#float)) – Refernce pressure in ATM.
- **temperature** (*float* (https://docs.python.org/3/library/functions.html#float)) – System temperature during observation in K
- **pressure** (*float* (https://docs.python.org/3/library/functions.html#float)) – System pressure during observation in ATM.

Attributes Summary

inputs
outputs
standard_broadcasting

Methods Summary

__call__(lam, alpha_in, beta_in, zin[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
compute_refraction_index(lam, temp, tref, ...)	Calculate and retrun the refraction index.
evaluate(lam, alpha_in, beta_in, zin)	Go through the prism

Attributes Documentation

inputs = ('lam', 'alpha_in', 'beta_in', 'zin')

outputs = ('alpha_out', 'beta_out', 'zout')

standard_broadcasting = False

Methods Documentation

__call__(*lam*, *alpha_in*, *beta_in*, *zin*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static compute_refraction_index(*lam*, *temp*, *tref*, *pref*, *pressure*, *kcoef*, *lcoef*, *tcoef*)
Calculate and retrun the refraction index.

evaluate(*lam*, *alpha_in*, *beta_in*, *zin*)
Go through the prism

Logical

class jwst.transforms.**Logical**(*condition*, *compareto*, *value*, ***kwargs*)
Bases: astropy.modeling.core.Model

Substitute values in an array where the condition is evaluated to True.

Similar to numpy's where function.

Parameters

- **condition** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A string representing the logical, one of GT, LT, NE, EQ
- **compareto** (*float* (https://docs.python.org/3/library/functions.html#float), *ndarray*) – A number to compare to using the condition If ndarray then the input array, compareto and value should have the same shape.
- **value** (*float* (https://docs.python.org/3/library/functions.html#float), *ndarray*) – Value to substitute where condition is True.

Attributes Summary

conditions	
inputs	
outputs	

Methods Summary

__call__(x[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x)	Evaluate the model on some input variables.

Attributes Documentation

conditions = {'EQ': <ufunc 'equal'>, 'GT': <ufunc 'greater'>, 'LT': <ufunc 'less'>, 'N

inputs = ('x',)

outputs = ('x',)

Methods Documentation

__call__ (*x*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate (*x*)
Evaluate the model on some input variables.

NirissSOSSModel

class jwst.transforms.**NirissSOSSModel** (*spectral_orders*, *models*)
Bases: astropy.modeling.core.Model

NIRISS SOSS wavelength solution implemented as a Model.

Parameters

- **spectral_orders** (*list of int*) – Spectral orders for which there is a wavelength solution.
- **models** (list of Model) – A list of transforms representing the wavelength solution for each order in spectral orders. It should match the order in spectral_orders.

Attributes Summary

inputs	x and y pixel coordinates and spectral order
outputs	RA and DEC coordinates and wavelength

Methods Summary

__call__(x, y, spectral_order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, spectral_order)	Evaluate the model on some input variables.
get_model(spectral_order)	

Attributes Documentation

inputs = ('x', 'y', 'spectral_order')
x and y pixel coordinates and spectral order

outputs = ('ra', 'dec', 'lam')
RA and DEC coordinates and wavelength

Methods Documentation

__call__(*x*, *y*, *spectral_order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *spectral_order*)
Evaluate the model on some input variables.

get_model(*spectral_order*)

V23ToSky

class jwst.transforms.**V23ToSky**(*angles*, *axes_order*, *name=None*)
Bases: jwst.transforms.models.Rotation3D

Transform from V2V3 to a standard coordinate system (ICRS).

Parameters

- **angles** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – A sequence of angles (in deg). The angles are [-V2_REF, V3_REF, -ROLL_REF, -DEC_REF, RA_REF].
- **axes_order** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A sequence of characters ('x', 'y', or 'z') corresponding to the axis of rotation and matching the order in `angles`. The axes are "zyxyz".

Attributes Summary

inputs	Coordinates in the (V2, V3) telescope frame.
outputs	RA, DEC cooridnates in ICRS.
param_names	

Methods Summary

__call__(v2, v3)	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
cartesian2spherical(x, y, z)	Convert cartesian coordinates to spherical coordinates (in deg).
evaluate(v2, v3, angles)	Apply the rotation to a set of 3D Cartesian coordinates.
spherical2cartesian(alpha, delta)	Convert spherical coordinates (in deg) to cartesian.

Attributes Documentation

inputs = ('v2', 'v3')
Coordinates in the (V2, V3) telescope frame.

Type ("v2", "v3")

outputs = ('ra', 'dec')
RA, DEC cooridnates in ICRS.

Type ("ra", "dec")

param_names = ('angles',)

Methods Documentation

__call__(*v2*, *v3*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static cartesian2spherical(*x*, *y*, *z*)
Convert cartesian coordinates to spherical coordinates (in deg).

evaluate(*v2*, *v3*, *angles*)
Apply the rotation to a set of 3D Cartesian coordinates.

static spherical2cartesian(*alpha*, *delta*)
Convert spherical coordinates (in deg) to cartesian.

Slit

class jwst.transforms.**Slit**(*name*, *shutter_id*, *xcen*, *ycen*, *ymin*, *ymax*, *quadrant*, *source_id*, *shutter_state*, *source_name*, *source_alias*, *stellarity*, *source_xpos*, *source_ypos*)
Bases: tuple (https://docs.python.org/3/library/stdtypes.html#tuple)

Nirspec Slit structure definition

NIRCAMForwardRowGrismDispersion

class jwst.transforms.**NIRCAMForwardRowGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *name=None*, *meta=None*)
Bases: astropy.modeling.core.Model

Return the transform from grism to image for the given spectral order.

Parameters

- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[int* (https://docs.python.org/3/library/functions.html#int)*]*) – List of orders which are available
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the wavelength solutions for each order
- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the x solutions for each order
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which givern the y solutions for each order

Returns

- *x, y, wavelength, order in the grism image for the pixel at x0,y0 that was*
- *specified as input using the input delta pix for the specified order*

Notes

The evaluation here is linear currently because higher orders have not yet been defined for NIRCAM (NIRCAM polynomials currently do not have any field dependence)

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the transform from grism to image for the given spectral order.

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'x0', 'y0', 'order')

linear = False

outputs = ('x', 'y', 'wavelength', 'order')

standard_broadcasting = False

Methods Documentation

__call__ (*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate (*x*, *y*, *x0*, *y0*, *order*)
Return the transform from grism to image for the given spectral order.

Parameters

- **x** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x pixel
- **y** (*float* (https://docs.python.org/3/library/functions.html#float)) – intput y pixel
- **x0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x-center of object
- **y0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input y-center of object
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – the spectral order to use

NIRCAMForwardColumnGrismDispersion

class jwst.transforms.**NIRCAMForwardColumnGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *name=None*, *meta=None*)

Bases: astropy.modeling.core.Model

Return the transform from grism to image for the given spectral order.

Parameters

- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[int* (https://docs.python.org/3/library/functions.html#int)*]*) – List of orders which are available
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the wavelength solutions
- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the x solutions
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which givern the y solutions

Returns

- *x, y, lam, order in the grism image for the pixel at x0,y0 that was*
- *specified as input using the input delta pix for the specified order*

Notes

The evaluation here is lineaer because higher orders have not yet been defined for NIRCAM (NIRCAM polynomials currently do not have any field dependence)

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the transform from grism to image for the given spectral order.

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'x0', 'y0', 'order')

linear = False

outputs = ('x', 'y', 'wavelength', 'order')

standard_broadcasting = False

Methods Documentation

__call__ (*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)

Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate (*x*, *y*, *x0*, *y0*, *order*)

Return the transform from grism to image for the given spectral order.

Parameters

- **x** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x pixel
- **y** (*float* (https://docs.python.org/3/library/functions.html#float)) – intput y pixel
- **x0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x-center of object
- **y0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input y-center of object
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – the spectral order to use

NIRCAMBackwardGrismDispersion

class jwst.transforms.**NIRCAMBackwardGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *name=None*, *meta=None*)

Bases: astropy.modeling.core.Model

Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[int* (https://docs.python.org/3/library/functions.html#int)*]*) – List of orders which are available
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the wavelength solutions
- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the x solutions
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which givern the y solutions

Returns

- *x, y, lam, order in the grism image for the pixel at x0,y0 that was*
- *specified as input using the wavelength l for the specified order*

Notes

The evaluation here is lineaer because higher orders have not yet been defined for NIRCAM (NIRCAM polynomials currently do not have any field dependence)

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, wavelength, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, wavelength, order)	Return the tranfrom from image to grism for the given spectral order.

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'wavelength', 'order')

linear = False

outputs = ('x', 'y', 'x0', 'y0', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *wavelength*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *wavelength*, *order*)
Return the tranfrom from image to grism for the given spectral order.

Parameters

- **x** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x pixel
- **y** (*float* (https://docs.python.org/3/library/functions.html#float)) – intput y pixel
- **wavelength** (*float* (https://docs.python.org/3/library/functions.html#float)) – input wavelength in angstroms
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – specifies the spectral order

MIRI_AB2Slice

class jwst.transforms.**MIRI_AB2Slice**(*beta_zero=0*, *beta_del=1*, *channel=1*, ***kwargs*)
Bases: astropy.modeling.core.Model

MIRI MRS alpha, beta to slice transform

Parameters

- **beta_zero** (*float* (https://docs.python.org/3/library/functions.html#float)) –
- **beta_del** (*float* (https://docs.python.org/3/library/functions.html#float)) –

Attributes Summary

beta_del	Beta_del parameter
beta_zero	Beta_zero parameter
channel	one of 1, 2, 3, 4
inputs	the beta angle
outputs	Slice number
param_names	
standard_broadcasting	

Methods Summary

__call__(beta[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(beta, beta_zero, beta_del, channel)	Evaluate the model on some input variables.

Attributes Documentation

beta_del
Beta_del parameter

beta_zero
Beta_zero parameter

channel
one of 1, 2, 3, 4

Type MIRI MRS channel

inputs = ('beta',)
the beta angle

Type "beta"

outputs = ('slice',)
Slice number

Type "slice"

param_names = ('beta_zero', 'beta_del', 'channel')

standard_broadcasting = False

Methods Documentation

__call__(*beta*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static evaluate(*beta*, *beta_zero*, *beta_del*, *channel*)
Evaluate the model on some input variables.

GrismObject

class jwst.transforms.**GrismObject**
Bases: *jwst.transforms.models.GrismObject*

Grism Objects identified from a direct image catalog and segment map.

Notes

The object bounding box is computed from the segementation map, using the min and max wavelegnth for each of the orders that are available. The order_bounding member is a dictionary of bounding boxes for the object keyed by order

ra and dec are the sky ra and dec of the center of the object as measured from the non-dispersed image.

the segment_[ra/dec][min/max] are also as measured on the direct image

order_bounding is stored as a lookup dictionary per order and contains the object x,y bounding location on the grism image GrismObject(order_bounding={"+1":((xmin,xmax),(ymin,ymax)),"+2":((2,3),(2,3))})

`sky_bbox_??` contains the ra,dec,frame information for the bbox from the catalog

NIRISSForwardRowGrismDispersion

class jwst.transforms.**NIRISSForwardRowGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *theta=0.0*, *name=None*, *meta=None*)

Bases: `astropy.modeling.core.Model`

This model calculates the dispersion extent of NIRISS pixels.

The dispersion polynomial is relative to the input x,y pixels in the direct image for a given wavelength.

Parameters

- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuples]*) – The list of tuple(models) for the polynomial model in x
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuples]*) – The list of tuple(models) for the polynomial model in y
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of models for the polynomial model in l
- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of orders which are available to the model

Notes

Given the x,y, source location as known on the dispersed image, as well as order, it returns the tuple of x,y,wavelength,order.

This model needs to be generalized, at the moment it satisfies the 2t x 6(xy)th order polynomial currently used by NIRISS.

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the valid pixel(s) and wavelengths given center x,y and lam

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'x0', 'y0', 'order')

linear = False

outputs = ('x', 'y', 'wavelength', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *x0*, *y0*, *order*)
Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **x0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Source object x-center
- **y0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Source object y-center
- **x** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Input x location
- **y** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Input y location
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – Spectral order to use

Returns

- *x, y, lambda, order, theta, in the direct image for the pixel that was*
- *specified as input using the wavelength l and spectral order*

Notes

There's spatial dependence for NIRISS as well as dependence on the filter wheel rotation during the exposure.

NIRISSForwardColumnGrismDispersion

class jwst.transforms.**NIRISSForwardColumnGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *theta=None*, *name=None*, *meta=None*)

Bases: astropy.modeling.core.Model

This model calculates the dispersion extent of NIRISS pixels.

The dispersion polynomial is relative to the input x,y pixels in the direct image for a given wavelength.

Parameters

- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple)*]*) – The list of tuple(models) for the polynomial model in x
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple)*]*) – The list of tuple(models) for the polynomial model in y
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of models for the polynomial model in l
- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of orders which are available to the model

Notes

Given the x,y, source location, order, it returns the tuple of x,y,wavelength,order on the dispersed image. It also requires FWCPOS from the image header, this is the filter wheel position in degrees.

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the valid pixel(s) and wavelengths given center x,y and lam

Attributes Documentation

fittable = False

```
inputs = ('x', 'y', 'x0', 'y0', 'order')
linear = False
outputs = ('x', 'y', 'wavelength', 'order')
standard_broadcasting = False
```

Methods Documentation

__call__(*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *x0*, *y0*, *order*)
Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **x0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Source object x-center
- **y0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Source object y-center
- **x** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input x location
- **y** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input y location
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – Spectral order to use
- **theta** (*float* (https://docs.python.org/3/library/functions.html#float)) – input filter wheel rotation angle in degrees

Returns

- *x, y, lambda, order, in the direct image for the pixel that was*
- *specified as input using the wavelength l and spectral order*

Notes

There's spatial dependence for NIRISS as well as rotation for the filter wheel

NIRISSBackwardGrismDispersion

class jwst.transforms.**NIRISSBackwardGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *theta=None*, *name=None*, *meta=None*)
Bases: astropy.modeling.core.Model

This model calculates the dispersion extent of NIRISS pixels.

The dispersion is relative to the input x,y for a given wavelength.

Parameters

- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple)*]*) – The list of tuple(models) for the polynomial model in x
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple)*]*) – The list of tuple(models) for the polynomial model in y
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of models for the polynomial model in l
- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of orders which are available to the model
- **theta** (*float* (https://docs.python.org/3/library/functions.html#float)) – The rotation to apply

Notes

Given the x,y, wave, order as known on the direct image, it returns the tuple of x, y, wave, order for that wave in the dispersed image.

This model needs to be generalized, at the moment it satisfies the 2t x 6(xy)th order polynomial currently used by NIRISS.

There's spatial dependence for NIRISS so the forward transform is iterative

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, wavelength, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, wavelength, order)	Return the valid pixel(s) and wavelengths given center x,y and lam

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'wavelength', 'order')

linear = False

outputs = ('x', 'y', 'x0', 'y0', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *wavelength*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *wavelength*, *order*)
Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **wavelength** (*float* (https://docs.python.org/3/library/functions.html#float)) – Input wavelength you want to know about, will be converted to float
- **x** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input x location
- **y** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input y location
- **wavelength** – Wavelength to disperse
- **order** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The order to use

Returns

- *x, y, wavelength, order in the grism image for the pixel at x,y that was*
- *specified as input using the wavelength and order specified*

Notes

There's spatial dependence for NIRISS so the forward transform dependes on x,y as well as the filter wheel rotation. Theta is usu. taken to be the different between fwcpos_ref in the specwcs reference file and fwcpos from the input image.

V2V3ToIdeal

class jwst.transforms.**V2V3ToIdeal**(*v3idlyangle*, *v2ref*, *v3ref*, *vparity*, *name='V2idl'*, ***kwargs*)
Bases: astropy.modeling.core.Model

Performs the transform from telescope V2,V3 to Ideal coordinate system. The two systems have the same origin - V2_REF, V3_REF.

Note: This model has no schema implemented - add if needed.

Attributes Summary

inputs	coorinates in the telescope (V2,V3) frame.
outputs	x and y coordinates in the telescope Ideal frame.
param_names	
v2ref	
v3idlyangle	
v3ref	
vparity	

Methods Summary

__call__(v2, v3[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(v2, v3, v3idlyangle, v2ref, v3ref, ...)	**param xidl, yidl** Coordinates in Ideal System [in arcsec]

Attributes Documentation

inputs = ('v2', 'v3')
coorinates in the telescope (V2,V3) frame.

Type ('v2', 'v3')

outputs = ('xidl', 'yidl')
x and y coordinates in the telescope Ideal frame.

Type ('xidl', 'yidl')

param_names = ('v3idlyangle', 'v2ref', 'v3ref', 'vparity')

v2ref

v3idlyangle

v3ref

vparity

Methods Documentation

__call__(*v2*, *v3*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static evaluate(*v2*, *v3*, *v3idlyangle*, *v2ref*, *v3ref*, *vparity*)

Parameters

- **yidl** (*xidl,*) – Coordinates in Ideal System [in arcsec]
- **v3idlyangle** (*float* (https://docs.python.org/3/library/functions.html#float)) – Angle between Ideal Y-axis and V3 [in deg]
- **v3ref** (*v2ref,*) – Coordinates in V2, V3 [in arcsec]
- **vparity** (*int* (https://docs.python.org/3/library/functions.html#int)) – Parity.

Returns **xidl, yidl** – Coordinates in the Ideal telescope system [in arcsec].

Return type ndarray-like

IdealToV2V3

class jwst.transforms.**IdealToV2V3**(*v3idlyangle*, *v2ref*, *v3ref*, *vparity*, *name='idl2V'*, ***kwargs*)
Bases: astropy.modeling.core.Model

Performs the transform from Ideal to telescope V2,V3 coordinate system. The two systems have the same origin: V2_REF, V3_REF.

Note: This model has no schema implemented - add schema if needed.

Attributes Summary

inputs	x and y coordinates in the telescope Ideal frame.
outputs	coorinates in the telescope (V2,V3) frame.
param_names	
v2ref	
v3idlyangle	
v3ref	
vparity	

Methods Summary

__call__(xidl, yidl[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(xidl, yidl, v3idlyangle, v2ref, ...)	**param xidl, yidl** Coordinates in Ideal System [in arcsec]

Attributes Documentation

inputs = ('xidl', 'yidl')
x and y coordinates in the telescope Ideal frame.

outputs = ('v2', 'v3')
coorinates in the telescope (V2,V3) frame.

param_names = ('v3idlyangle', 'v2ref', 'v3ref', 'vparity')

v2ref

v3idlyangle

v3ref

vparity

Methods Documentation

__call__(*xidl*, *yidl*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static evaluate(*xidl*, *yidl*, *v3idlyangle*, *v2ref*, *v3ref*, *vparity*)

Parameters

- **yidl** (*xidl,*) – Coordinates in Ideal System [in arcsec]
- **v3idlyangle** (*float* (https://docs.python.org/3/library/functions.html#float)) – Angle between Ideal Y-axis and V3 [in deg]
- **v3ref** (*v2ref,*) – Coordinates in V2, V3 [in arcsec]
- **vparity** (*int* (https://docs.python.org/3/library/functions.html#int)) – Parity.

Returns **v2, v3** – Coordinates in the (V2, V3) telescope system [in arcsec].

Return type ndarray-like

IncompatibleCorrections

exception jwst.transforms.**IncompatibleCorrections**

An exception class used to report cases when two or more tangent plane corrections cannot be combined into a single one.

TPCorr

class jwst.transforms.**TPCorr**(*v2ref=0.0, v3ref=0.0, roll=0.0, matrix=[[1.0, 0.0], [0.0, 1.0]], shift=[0.0, 0.0], **kwargs*)

Bases: `astropy.modeling.core.Model`

Apply `V2ref`, `V3ref`, and `roll` to input angles and project the point from the tangent plane onto a celestial sphere.

Parameters

- **v2ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – V2 position of the reference point in degrees. Default is 0 degrees.
- **v3ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – V3 position of the reference point in degrees. Default is 0 degrees.
- **roll** (*float* (https://docs.python.org/3/library/functions.html#float)) – Roll angle in degrees. Default is 0 degrees.

Attributes Summary

input_units	This property is used to indicate what units or sets of units the evaluate method expects, and returns a dictionary mapping inputs to units (or `None` (https://docs.python.org/3/library/constants.html#None) if any units are accepted).
inputs	
matrix	
outputs	
param_names	
r0	Radius of the generating sphere.

Continued on next page

Table 388 – continued from previous page

return_units	This property is used to indicate what units or sets of units the output of evaluate should be in, and returns a dictionary mapping outputs to units (or None (https://docs.python.org/3/library/constants.html#None) if any units are accepted).
roll	
shift	
standard_broadcasting	
v2ref	
v3ref	

Methods Summary

__call__(v2, v3[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
cartesian2spherical(x, y, z)	Convert cartesian coordinates to spherical coordinates (in acrsec).
combine(t2, t1)	Combine transformation `t2` with another transformation (`t1`) *previously applied* to the coordinates.
evaluate(v2, v3, v2ref, v3ref, roll, matrix, ...)	Evaluate the model on some input variables.
spherical2cartesian(alpha, delta)	Convert spherical coordinates (in arcsec) to cartesian.
tanp_to_v2v3(xt, yt)	Converts tangent plane coordinates to V2V3 spherical coordinates.
v2v3_to_tanp(v2, v3)	Converts V2V3 spherical coordinates to tangent plane coordinates.

Attributes Documentation

input_units
: This property is used to indicate what units or sets of units the evaluate method expects, and returns a dictionary mapping inputs to units (or None (https://docs.python.org/3/library/constants.html#None) if any units are accepted).

  Model sub-classes can also use function annotations in evaluate to indicate valid input units, in which case this property should not be overriden since it will return the input units based on the annotations.

inputs = ('v2', 'v3')

matrix

outputs = ('v2c', 'v3c')

param_names = ('v2ref', 'v3ref', 'roll', 'matrix', 'shift')

r0 = 206264.80624709636
: Radius of the generating sphere. This sets the circumference to 360 deg so that arc length is measured in deg.

return_units
: This property is used to indicate what units or sets of units the output of evaluate should be in, and returns a dictionary mapping outputs to units (or None (https://docs.python.org/3/library/constants.html#None) if any units are accepted).

Model sub-classes can also use function annotations in evaluate to indicate valid output units, in which case this property should not be overriden since it will return the return units based on the annotations.

roll

shift

standard_broadcasting = False

v2ref

v3ref

Methods Documentation

__call__(*v2*, *v3*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static cartesian2spherical(*x*, *y*, *z*)
Convert cartesian coordinates to spherical coordinates (in acrsec).

classmethod combine(*t2*, *t1*)
Combine transformation `t2` with another transformation (`t1`) *previously applied* to the coordinates. That is, transformation `t2` is assumed to *follow* (=applied after) the transformation provided by the argument `t1`.

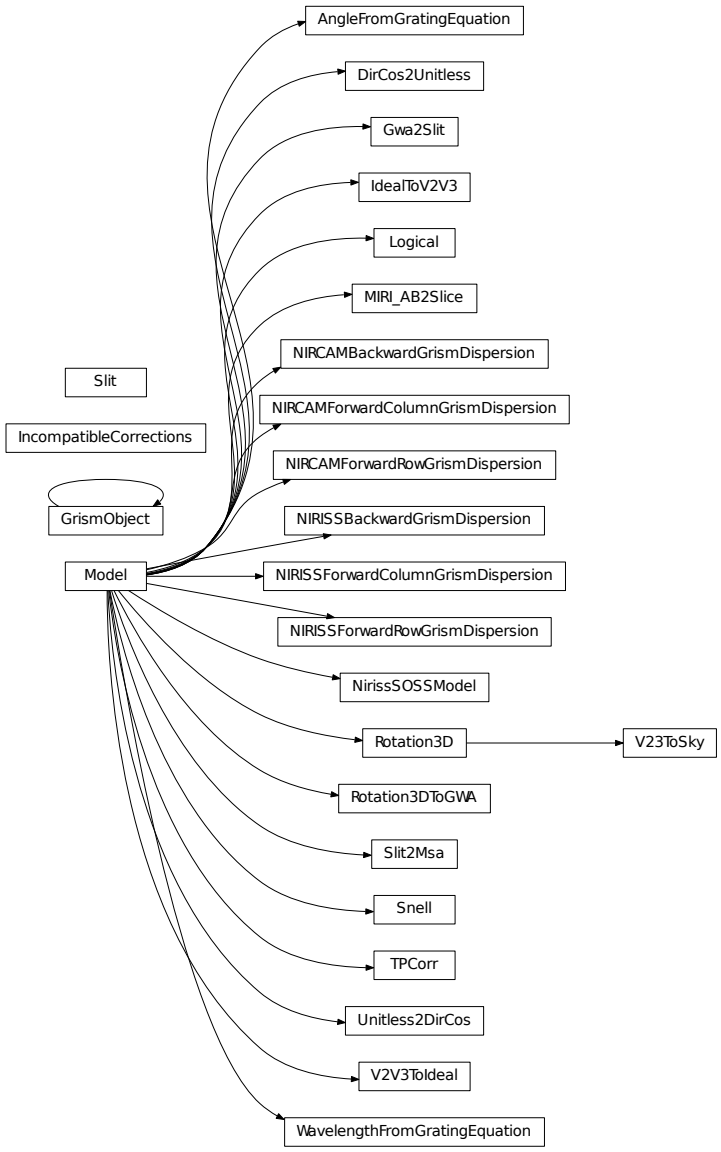
evaluate(*v2*, *v3*, *v2ref*, *v3ref*, *roll*, *matrix*, *shift*)
Evaluate the model on some input variables.

static spherical2cartesian(*alpha*, *delta*)
Convert spherical coordinates (in arcsec) to cartesian.

tanp_to_v2v3(*xt*, *yt*)
Converts tangent plane coordinates to V2V3 spherical coordinates.

v2v3_to_tanp(*v2*, *v3*)
Converts V2V3 spherical coordinates to tangent plane coordinates.

Class Inheritance Diagram



jwst.transforms.models Module

Models used by the JWST pipeline.

The models are written using the astropy.modeling framework. Since they are specific to JWST, the models and their ASDF schemas are kept here separately from astropy. An ASDF extension for this package is registered with ASDF through entry points.

Classes

AngleFromGratingEquation(groove_density, ...)	Solve the 3D Grating Dispersion Law for the refracted angle.
WavelengthFromGratingEquation(...)	Solve the 3D Grating Dispersion Law for the wavelength.
Unitless2DirCos(*args[, meta, name])	Transform a vector to directional cosines.
DirCos2Unitless(*args[, meta, name])	Transform directional cosines to vector.
Rotation3DToGWA(angles, axes_order[, name])	Perform a 3D rotation given an angle in degrees.
Gwa2Slit(slits, models)	NIRSpec GWA to slit transform.
Slit2Msa(slits, models)	NIRSpec slit to MSA transform.
Snell(angle, kcoef, lcoef, tcoef, tref, ...)	Apply transforms, including Snell law, through the NIRSpec prism.
Logical(condition, compareto, value, **kwargs)	Substitute values in an array where the condition is evaluated to True.
NirissSOSSModel(spectral_orders, models)	NIRISS SOSS wavelength solution implemented as a Model.
V23ToSky(angles, axes_order[, name])	Transform from V2V3 to a standard coordinate system (ICRS).
Slit(name, shutter_id, xcen, ycen, ymin, ...)	Nirspec Slit structure definition
NIRCAMForwardRowGrismDispersion(orders[, ...])	Return the transform from grism to image for the given spectral order.
NIRCAMForwardColumnGrismDispersion(orders[, ...])	Return the transform from grism to image for the given spectral order.
NIRCAMBackwardGrismDispersion(orders[, ...])	Return the valid pixel(s) and wavelengths given center x,y and lam
MIRI_AB2Slice([beta_zero, beta_del, channel])	MIRI MRS alpha, beta to slice transform
GrismObject	Grism Objects identified from a direct image catalog and segment map.
NIRISSForwardRowGrismDispersion(orders[, ...])	This model calculates the dispersion extent of NIRISS pixels.
NIRISSForwardColumnGrismDispersion(orders[, ...])	This model calculates the dispersion extent of NIRISS pixels.
NIRISSBackwardGrismDispersion(orders[, ...])	This model calculates the dispersion extent of NIRISS pixels.
V2V3ToIdeal(v3idlyangle, v2ref, v3ref, vparity)	Performs the transform from telescope V2,V3 to Ideal coordinate system.
IdealToV2V3(v3idlyangle, v2ref, v3ref, vparity)	Performs the transform from Ideal to telescope V2,V3 coordinate system.

AngleFromGratingEquation

class jwst.transforms.models.**AngleFromGratingEquation**(*groove_density*, *order*, ***kwargs*)

Bases: astropy.modeling.core.Model

Solve the 3D Grating Dispersion Law for the refracted angle.

Parameters

- **groove_density** (*int* (https://docs.python.org/3/library/functions.html#int)) – Grating ruling density.
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – Spectral order.

Attributes Summary

groove_density	Grating ruling density.
inputs	Wavelength and 3 angle coordinates going into the grating.
order	Spectral order.
outputs	Three angles coming out of the grating.
param_names	

Methods Summary

__call__(lam, alpha_in, beta_in, z[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(lam, alpha_in, beta_in, z, ...)	Evaluate the model on some input variables.

Attributes Documentation

groove_density
Grating ruling density.

inputs = ('lam', 'alpha_in', 'beta_in', 'z')
Wavelength and 3 angle coordinates going into the grating.

order
Spectral order.

outputs = ('alpha_out', 'beta_out', 'zout')
Three angles coming out of the grating.

param_names = ('groove_density', 'order')

Methods Documentation

__call__(*lam*, *alpha_in*, *beta_in*, *z*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*lam*, *alpha_in*, *beta_in*, *z*, *groove_density*, *order*)
Evaluate the model on some input variables.

WavelengthFromGratingEquation

class jwst.transforms.models.**WavelengthFromGratingEquation**(*groove_density*, *order*, ***kwargs*)

Bases: astropy.modeling.core.Model

Solve the 3D Grating Dispersion Law for the wavelength.

Parameters

- **groove_density** (*int* (https://docs.python.org/3/library/functions.html#int)) – Grating ruling density.

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

Attributes Summary

groove_density	Grating ruling density.
inputs	three angle - alpha_in and beta_in going into the grating and alpha_out coming out of the grating.
order	Spectral order.
outputs	Wavelength.
param_names	

Methods Summary

__call__(alpha_in, beta_in, alpha_out[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(alpha_in, beta_in, alpha_out, ...)	Evaluate the model on some input variables.

Attributes Documentation

groove_density
Grating ruling density.

inputs = ('alpha_in', 'beta_in', 'alpha_out')
three angle - alpha_in and beta_in going into the grating and alpha_out coming out of the grating.

order
Spectral order.

outputs = ('lam',)
Wavelength.

param_names = ('groove_density', 'order')

Methods Documentation

__call__(*alpha_in*, *beta_in*, *alpha_out*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*alpha_in*, *beta_in*, *alpha_out*, *groove_density*, *order*)
Evaluate the model on some input variables.

Unitless2DirCos

class jwst.transforms.models.**Unitless2DirCos**(**args*, *meta=None*, *name=None*, ***kwargs*)

Bases: astropy.modeling.core.Model

Transform a vector to directional cosines.

Attributes Summary

inputs	
outputs	

Methods Summary

__call__(x, y[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y)	Evaluate the model on some input variables.

Attributes Documentation

inputs = ('x', 'y')

outputs = ('x', 'y', 'z')

Methods Documentation

__call__(*x*, *y*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*)
Evaluate the model on some input variables.

DirCos2Unitless

class jwst.transforms.models.**DirCos2Unitless**(**args*, *meta=None*, *name=None*, ***kwargs*)

Bases: astropy.modeling.core.Model

Transform directional cosines to vector.

Attributes Summary

inputs	
outputs	

Methods Summary

__call__(x, y, z[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, z)	Evaluate the model on some input variables.

Attributes Documentation

inputs = ('x', 'y', 'z')

outputs = ('x', 'y')

Methods Documentation

__call__(*x*, *y*, *z*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *z*)
Evaluate the model on some input variables.

Rotation3DToGWA

class jwst.transforms.models.**Rotation3DToGWA**(*angles*, *axes_order*, *name=None*)
Bases: astropy.modeling.core.Model

Perform a 3D rotation given an angle in degrees.

Positive angles represent a counter-clockwise rotation and vice-versa.

Parameters

- **angles** (*array-like*) – Angles of rotation in deg in the order of axes_order.
- **axes_order** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A sequence of 'x', 'y', 'z' corresponding of axis of rotation/

Attributes Summary

angles
inputs
outputs
param_names
separable
standard_broadcasting

Methods Summary

__call__(x, y, z[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, z, angles)	Apply the rotation to a set of 3D Cartesian coordinates.

Attributes Documentation

angles

```
inputs = ('x', 'y', 'z')

outputs = ('x', 'y', 'z')

param_names = ('angles',)

separable = False

standard_broadcasting = False
```

Methods Documentation

__call__(*x*, *y*, *z*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *z*, *angles*)
Apply the rotation to a set of 3D Cartesian coordinates.

Gwa2Slit

class jwst.transforms.models.**Gwa2Slit**(*slits*, *models*)
Bases: astropy.modeling.core.Model

NIRSpec GWA to slit transform.

Parameters

- **slits** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – A list of open slits. A slit is a namedtupe of type *Slit* Slit("name", "shutter_id", "xcen", "ycen", "ymin", "ymax", "quadrant", "source_id", "shutter_state", "source_name", "source_alias", "stellarity", "source_xpos", "source_ypos"])
- **models** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – List of models (Model) corresponding to the list of slits.

Attributes Summary

inputs	Name of the slit and the three angle coordinates at the GWA going from detector to sky.
outputs	Name of the slit, x and y coordinates within the virtual slit and wavelength.
slits	

Methods Summary

__call__(name, angle1, angle2, angle3[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(name, x, y, z)	Evaluate the model on some input variables.
get_model(name)	

Attributes Documentation

inputs = ('name', 'angle1', 'angle2', 'angle3')
Name of the slit and the three angle coordinates at the GWA going from detector to sky.

outputs = ('name', 'x_slit', 'y_slit', 'lam')
Name of the slit, x and y coordinates within the virtual slit and wavelength.

slits

Methods Documentation

__call__(*name*, *angle1*, *angle2*, *angle3*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*name*, *x*, *y*, *z*)
Evaluate the model on some input variables.

get_model(*name*)

Slit2Msa

class jwst.transforms.models.**Slit2Msa**(*slits*, *models*)
Bases: astropy.modeling.core.Model

NIRSpec slit to MSA transform.

Parameters

- **slits** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – A list of open slits. A slit is a namedtupe, *Slit* Slit("name", "shutter_id", "xcen", "ycen", "ymin", "ymax", "quadrant", "source_id", "shutter_state", "source_name", "source_alias", "stellarity", "source_xpos", "source_ypos")
- **models** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – List of models (Model) corresponding to the list of slits.

Attributes Summary

inputs	Name of the slit, x and y coordinates within the virtual slit.
outputs	x and y coordinates in the MSA frame.
slits	

Methods Summary

__call__(name, x_slit, y_slit[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(name, x, y)	Evaluate the model on some input variables.
get_model(name)	

Attributes Documentation

inputs = ('name', 'x_slit', 'y_slit')
Name of the slit, x and y coordinates within the virtual slit.

outputs = ('x_msa', 'y_msa')
x and y coordinates in the MSA frame.

slits

Methods Documentation

__call__(*name*, *x_slit*, *y_slit*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*name*, *x*, *y*)
Evaluate the model on some input variables.

get_model(*name*)

Snell

class jwst.transforms.models.**Snell**(*angle*, *kcoef*, *lcoef*, *tcoef*, *tref*, *pref*, *temperature*, *pressure*, *name=None*)
Bases: astropy.modeling.core.Model

Apply transforms, including Snell law, through the NIRSpec prism.

Parameters

- **angle** (*flaot*) – Prism angle in deg.
- **kcoef** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – K coefficients in Sellmeir equation.
- **lcoef** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – L coefficients in Sellmeir equation.
- **tcoef** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – Thermal coefficients of glass.
- **tref** (*float* (https://docs.python.org/3/library/functions.html#float)) – Refernce temperature in K.
- **pref** (*float* (https://docs.python.org/3/library/functions.html#float)) – Refernce pressure in ATM.
- **temperature** (*float* (https://docs.python.org/3/library/functions.html#float)) – System temperature during observation in K
- **pressure** (*float* (https://docs.python.org/3/library/functions.html#float)) – System pressure during observation in ATM.

Attributes Summary

inputs
outputs
standard_broadcasting

Methods Summary

__call__(lam, alpha_in, beta_in, zin[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
compute_refraction_index(lam, temp, tref, ...)	Calculate and retrun the refraction index.
evaluate(lam, alpha_in, beta_in, zin)	Go through the prism

Attributes Documentation

inputs = ('lam', 'alpha_in', 'beta_in', 'zin')

outputs = ('alpha_out', 'beta_out', 'zout')

standard_broadcasting = False

Methods Documentation

__call__(*lam*, *alpha_in*, *beta_in*, *zin*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static compute_refraction_index(*lam*, *temp*, *tref*, *pref*, *pressure*, *kcoef*, *lcoef*, *tcoef*)
Calculate and retrun the refraction index.

evaluate(*lam*, *alpha_in*, *beta_in*, *zin*)
Go through the prism

Logical

class jwst.transforms.models.**Logical**(*condition*, *compareto*, *value*, ***kwargs*)
Bases: astropy.modeling.core.Model

Substitute values in an array where the condition is evaluated to True.

Similar to numpy's where function.

Parameters

- **condition** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A string representing the logical, one of GT, LT, NE, EQ
- **compareto** (*float* (https://docs.python.org/3/library/functions.html#float), *ndarray*) – A number to compare to using the condition If ndarray then the input array, compareto and value should have the same shape.
- **value** (*float* (https://docs.python.org/3/library/functions.html#float), *ndarray*) – Value to substitute where condition is True.

Attributes Summary

conditions	
inputs	
outputs	

Methods Summary

__call__(x[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x)	Evaluate the model on some input variables.

Attributes Documentation

conditions = {'EQ': <ufunc 'equal'>, 'GT': <ufunc 'greater'>, 'LT': <ufunc 'less'>, 'N

inputs = ('x',)

outputs = ('x',)

Methods Documentation

__call__ (*x*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate (*x*)
Evaluate the model on some input variables.

NirissSOSSModel

class jwst.transforms.models.**NirissSOSSModel** (*spectral_orders*, *models*)
Bases: astropy.modeling.core.Model

NIRISS SOSS wavelength solution implemented as a Model.

Parameters

- **spectral_orders** (*list of int*) – Spectral orders for which there is a wavelength solution.
- **models** (list of Model) – A list of transforms representing the wavelength solution for each order in spectral orders. It should match the order in spectral_orders.

Attributes Summary

inputs	x and y pixel coordinates and spectral order
outputs	RA and DEC coordinates and wavelength

Methods Summary

__call__(x, y, spectral_order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, spectral_order)	Evaluate the model on some input variables.
get_model(spectral_order)	

Attributes Documentation

inputs = ('x', 'y', 'spectral_order')
x and y pixel coordinates and spectral order

outputs = ('ra', 'dec', 'lam')
RA and DEC coordinates and wavelength

Methods Documentation

__call__(*x*, *y*, *spectral_order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *spectral_order*)
Evaluate the model on some input variables.

get_model(*spectral_order*)

V23ToSky

class jwst.transforms.models.**V23ToSky**(*angles*, *axes_order*, *name=None*)
Bases: jwst.transforms.models.Rotation3D

Transform from V2V3 to a standard coordinate system (ICRS).

Parameters

- **angles** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – A sequence of angles (in deg). The angles are [-V2_REF, V3_REF, -ROLL_REF, -DEC_REF, RA_REF].
- **axes_order** (*str* (https://docs.python.org/3/library/stdtypes.html#str)) – A sequence of characters ('x', 'y', or 'z') corresponding to the axis of rotation and matching the order in `angles`. The axes are "zyxyz".

Attributes Summary

inputs	Coordinates in the (V2, V3) telescope frame.
outputs	RA, DEC cooridnates in ICRS.
param_names	

Methods Summary

__call__(v2, v3)	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
cartesian2spherical(x, y, z)	Convert cartesian coordinates to spherical coordinates (in deg).
evaluate(v2, v3, angles)	Apply the rotation to a set of 3D Cartesian coordinates.
spherical2cartesian(alpha, delta)	Convert spherical coordinates (in deg) to cartesian.

Attributes Documentation

inputs = ('v2', 'v3')
Coordinates in the (V2, V3) telescope frame.

Type ("v2", "v3")

outputs = ('ra', 'dec')
RA, DEC cooridnates in ICRS.

Type ("ra", "dec")

param_names = ('angles',)

Methods Documentation

__call__(*v2*, *v3*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static cartesian2spherical(*x*, *y*, *z*)
Convert cartesian coordinates to spherical coordinates (in deg).

evaluate(*v2*, *v3*, *angles*)
Apply the rotation to a set of 3D Cartesian coordinates.

static spherical2cartesian(*alpha*, *delta*)
Convert spherical coordinates (in deg) to cartesian.

Slit

class jwst.transforms.models.**Slit**(*name*, *shutter_id*, *xcen*, *ycen*, *ymin*, *ymax*, *quadrant*, *source_id*, *shutter_state*, *source_name*, *source_alias*, *stellarity*, *source_xpos*, *source_ypos*)
Bases: tuple (https://docs.python.org/3/library/stdtypes.html#tuple)

Nirspec Slit structure definition

NIRCAMForwardRowGrismDispersion

class jwst.transforms.models.**NIRCAMForwardRowGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *name=None*, *meta=None*)

Bases: `astropy.modeling.core.Model`

Return the transform from grism to image for the given spectral order.

Parameters

- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[int* (https://docs.python.org/3/library/functions.html#int)*]*) – List of orders which are available
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the wavelength solutions for each order
- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the x solutions for each order
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which givern the y solutions for each order

Returns

- *x, y, wavelength, order in the grism image for the pixel at x0,y0 that was*
- *specified as input using the input delta pix for the specified order*

Notes

The evaluation here is linear currently because higher orders have not yet been defined for NIRCAM (NIRCAM polynomials currently do not have any field dependence)

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the transform from grism to image for the given spectral order.

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'x0', 'y0', 'order')

linear = False

outputs = ('x', 'y', 'wavelength', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *x0*, *y0*, *order*)
Return the transform from grism to image for the given spectral order.

Parameters

- **x** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x pixel
- **y** (*float* (https://docs.python.org/3/library/functions.html#float)) – intput y pixel
- **x0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x-center of object
- **y0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input y-center of object
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – the spectral order to use

NIRCAMForwardColumnGrismDispersion

class jwst.transforms.models.**NIRCAMForwardColumnGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *name=None*, *meta=None*)

Bases: astropy.modeling.core.Model

Return the transform from grism to image for the given spectral order.

Parameters

- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[int* (https://docs.python.org/3/library/functions.html#int)*]*) – List of orders which are available
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the wavelength solutions
- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the x solutions

- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which givern the y solutions

Returns

- *x, y, lam, order in the grism image for the pixel at x0,y0 that was*
- *specified as input using the input delta pix for the specified order*

Notes

The evaluation here is lineaer because higher orders have not yet been defined for NIRCAM (NIRCAM polynomials currently do not have any field dependence)

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the transform from grism to image for the given spectral order.

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'x0', 'y0', 'order')

linear = False

outputs = ('x', 'y', 'wavelength', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *x0*, *y0*, *order*)
Return the transform from grism to image for the given spectral order.

Parameters

- **x** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x pixel

- **y** (*float* (https://docs.python.org/3/library/functions.html#float)) – intput y pixel
- **x0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x-center of object
- **y0** (*float* (https://docs.python.org/3/library/functions.html#float)) – input y-center of object
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – the spectral order to use

NIRCAMBackwardGrismDispersion

class jwst.transforms.models.**NIRCAMBackwardGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *name=None*, *meta=None*)

Bases: astropy.modeling.core.Model

Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[int* (https://docs.python.org/3/library/functions.html#int)*]*) – List of orders which are available
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the wavelength solutions
- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which govern the x solutions
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[astropy.modeling.Model]*) – List of models which givern the y solutions

Returns

- *x, y, lam, order in the grism image for the pixel at x0,y0 that was*
- *specified as input using the wavelength l for the specified order*

Notes

The evaluation here is lineaer because higher orders have not yet been defined for NIRCAM (NIRCAM polynomials currently do not have any field dependence)

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, wavelength, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, wavelength, order)	Return the tranfrom from image to grism for the given spectral order.

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'wavelength', 'order')

linear = False

outputs = ('x', 'y', 'x0', 'y0', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *wavelength*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)

Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *wavelength*, *order*)

Return the tranfrom from image to grism for the given spectral order.

Parameters

- **x** (*float* (https://docs.python.org/3/library/functions.html#float)) – input x pixel
- **y** (*float* (https://docs.python.org/3/library/functions.html#float)) – intput y pixel
- **wavelength** (*float* (https://docs.python.org/3/library/functions.html#float)) – input wavelength in angstroms
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – specifies the spectral order

MIRI_AB2Slice

class jwst.transforms.models.**MIRI_AB2Slice**(*beta_zero=0*, *beta_del=1*, *channel=1*, ***kwargs*)

Bases: astropy.modeling.core.Model

MIRI MRS alpha, beta to slice transform

Parameters

- **beta_zero** (*float* (https://docs.python.org/3/library/functions.html#float)) –
- **beta_del** (*float* (https://docs.python.org/3/library/functions.html#float)) –

Attributes Summary

beta_del	Beta_del parameter
beta_zero	Beta_zero parameter
channel	one of 1, 2, 3, 4
inputs	the beta angle
outputs	Slice number
param_names	
standard_broadcasting	

Methods Summary

__call__(beta[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(beta, beta_zero, beta_del, channel)	Evaluate the model on some input variables.

Attributes Documentation

beta_del
Beta_del parameter

beta_zero
Beta_zero parameter

channel
one of 1, 2, 3, 4

Type MIRI MRS channel

inputs = ('beta',)
the beta angle

Type "beta"

outputs = ('slice',)
Slice number

Type "slice"

param_names = ('beta_zero', 'beta_del', 'channel')

standard_broadcasting = False

Methods Documentation

__call__(*beta*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static evaluate(*beta*, *beta_zero*, *beta_del*, *channel*)
Evaluate the model on some input variables.

GrismObject

class jwst.transforms.models.**GrismObject**
Bases: *jwst.transforms.models.GrismObject*

Grism Objects identified from a direct image catalog and segment map.

Notes

The object bounding box is computed from the segementation map, using the min and max wavelegnth for each of the orders that are available. The order_bounding member is a dictionary of bounding boxes for the object keyed by order

ra and dec are the sky ra and dec of the center of the object as measured from the non-dispersed image.

the segment_[ra/dec][min/max] are also as measured on the direct image

order_bounding is stored as a lookup dictionary per order and contains the object x,y bounding location on the grism image GrismObject(order_bounding={"+1":((xmin,xmax),(ymin,ymax)),"+2":((2,3),(2,3))})

`sky_bbox_??` contains the ra,dec,frame information for the bbox from the catalog

NIRISSForwardRowGrismDispersion

class jwst.transforms.models.**NIRISSForwardRowGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *theta=0.0*, *name=None*, *meta=None*)
Bases: `astropy.modeling.core.Model`

This model calculates the dispersion extent of NIRISS pixels.

The dispersion polynomial is relative to the input x,y pixels in the direct image for a given wavelength.

Parameters

- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuples]*) – The list of tuple(models) for the polynomial model in x
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuples]*) – The list of tuple(models) for the polynomial model in y
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of models for the polynomial model in l
- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of orders which are available to the model

Notes

Given the x,y, source location as known on the dispersed image, as well as order, it returns the tuple of x,y,wavelength,order.

This model needs to be generalized, at the moment it satisfies the 2t x 6(xy)th order polynomial currently used by NIRISS.

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the valid pixel(s) and wavelengths given center x,y and lam

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'x0', 'y0', 'order')

linear = False

outputs = ('x', 'y', 'wavelength', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)

Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *x0*, *y0*, *order*)

Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **x0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Source object x-center
- **y0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Source object y-center
- **x** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Input x location
- **y** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float), *list* (https://docs.python.org/3/library/stdtypes.html#list)) – Input y location

- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – Spectral order to use

Returns

- *x, y, lambda, order, theta, in the direct image for the pixel that was*
- *specified as input using the wavelength l and spectral order*

Notes

There's spatial dependence for NIRISS as well as dependence on the filter wheel rotation during the exposure.

NIRISSForwardColumnGrismDispersion

class jwst.transforms.models.**NIRISSForwardColumnGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *theta=None*, *name=None*, *meta=None*)

Bases: `astropy.modeling.core.Model`

This model calculates the dispersion extent of NIRISS pixels.

The dispersion polynomial is relative to the input x,y pixels in the direct image for a given wavelength.

Parameters

- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple) *]*) – The list of tuple(models) for the polynomial model in x
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple) *]*) – The list of tuple(models) for the polynomial model in y
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of models for the polynomial model in l
- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of orders which are available to the model

Notes

Given the x,y, source location, order, it returns the tuple of x,y,wavelength,order on the dispersed image. It also requires FWCPOS from the image header, this is the filter wheel position in degrees.

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, x0, y0, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, x0, y0, order)	Return the valid pixel(s) and wavelengths given center x,y and lam

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'x0', 'y0', 'order')

linear = False

outputs = ('x', 'y', 'wavelength', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *x0*, *y0*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *x0*, *y0*, *order*)
Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **x0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Source object x-center
- **y0** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Source object y-center
- **x** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input x location
- **y** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input y location
- **order** (*int* (https://docs.python.org/3/library/functions.html#int)) – Spectral order to use
- **theta** (*float* (https://docs.python.org/3/library/functions.html#float)) – input filter wheel rotation angle in degrees

Returns

- *x, y, lambda, order, in the direct image for the pixel that was*
- *specified as input using the wavelength l and spectral order*

Notes

There's spatial dependence for NIRISS as well as rotation for the filter wheel

NIRISSBackwardGrismDispersion

class jwst.transforms.models.**NIRISSBackwardGrismDispersion**(*orders*, *lmodels=None*, *xmodels=None*, *ymodels=None*, *theta=None*, *name=None*, *meta=None*)

Bases: astropy.modeling.core.Model

This model calculates the dispersion extent of NIRISS pixels.

The dispersion is relative to the input x,y for a given wavelength.

Parameters

- **xmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple) *]*) – The list of tuple(models) for the polynomial model in x
- **ymodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list) *[tuple* (https://docs.python.org/3/library/stdtypes.html#tuple) *]*) – The list of tuple(models) for the polynomial model in y
- **lmodels** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of models for the polynomial model in l
- **orders** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The list of orders which are available to the model
- **theta** (*float* (https://docs.python.org/3/library/functions.html#float)) – The rotation to apply

Notes

Given the x,y, wave, order as known on the direct image, it returns the tuple of x, y, wave, order for that wave in the dispersed image.

This model needs to be generalized, at the moment it satisfies the 2t x 6(xy)th order polynomial currently used by NIRISS.

There's spatial dependence for NIRISS so the forward transform is iterative

Attributes Summary

fittable
inputs
linear
outputs
standard_broadcasting

Methods Summary

__call__(x, y, wavelength, order[, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(x, y, wavelength, order)	Return the valid pixel(s) and wavelengths given center x,y and lam

Attributes Documentation

fittable = False

inputs = ('x', 'y', 'wavelength', 'order')

linear = False

outputs = ('x', 'y', 'x0', 'y0', 'order')

standard_broadcasting = False

Methods Documentation

__call__(*x*, *y*, *wavelength*, *order*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)

Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

evaluate(*x*, *y*, *wavelength*, *order*)

Return the valid pixel(s) and wavelengths given center x,y and lam

Parameters

- **wavelength** (*float* (https://docs.python.org/3/library/functions.html#float)) – Input wavelength you want to know about, will be converted to float
- **x** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input x location
- **y** (*int* (https://docs.python.org/3/library/functions.html#int), *float* (https://docs.python.org/3/library/functions.html#float)) – Input y location
- **wavelength** – Wavelength to disperse
- **order** (*list* (https://docs.python.org/3/library/stdtypes.html#list)) – The order to use

Returns

- *x, y, wavelength, order in the grism image for the pixel at x,y that was*
- *specified as input using the wavelength and order specified*

Notes

There's spatial dependence for NIRISS so the forward transform dependes on x,y as well as the filter wheel rotation. Theta is usu. taken to be the different between fwcpos_ref in the specwcs reference file and fwcpos from the input image.

V2V3ToIdeal

class jwst.transforms.models.**V2V3ToIdeal**(*v3idlyangle*, *v2ref*, *v3ref*, *vparity*, *name='V2idl'*, ***kwargs*)

Bases: astropy.modeling.core.Model

Performs the transform from telescope V2,V3 to Ideal coordinate system. The two systems have the same origin - V2_REF, V3_REF.

Note: This model has no schema implemented - add if needed.

Attributes Summary

inputs	coorinates in the telescope (V2,V3) frame.
outputs	x and y coordinates in the telescope Ideal frame.
param_names	
v2ref	
v3idlyangle	
v3ref	
vparity	

Methods Summary

__call__(v2, v3[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.
evaluate(v2, v3, v3idlyangle, v2ref, v3ref, ...)	**param xidl, yidl** Coordinates in Ideal System [in arcsec]

Attributes Documentation

inputs = ('v2', 'v3')
coorinates in the telescope (V2,V3) frame.

Type ('v2', 'v3')

outputs = ('xidl', 'yidl')
x and y coordinates in the telescope Ideal frame.

Type ('xidl', 'yidl')

param_names = ('v3idlyangle', 'v2ref', 'v3ref', 'vparity')

v2ref

v3idlyangle

v3ref

vparity

Methods Documentation

__call__(*v2*, *v3*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)

Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static evaluate(*v2*, *v3*, *v3idlyangle*, *v2ref*, *v3ref*, *vparity*)

Parameters

- **yidl** (*xidl,*) – Coordinates in Ideal System [in arcsec]
- **v3idlyangle** (*float* (https://docs.python.org/3/library/functions.html#float)) – Angle between Ideal Y-axis and V3 [in deg]
- **v3ref** (*v2ref,*) – Coordinates in V2, V3 [in arcsec]
- **vparity** (*int* (https://docs.python.org/3/library/functions.html#int)) – Parity.

Returns **xidl, yidl** – Coordinates in the Ideal telescope system [in arcsec].

Return type ndarray-like

IdealToV2V3

class jwst.transforms.models.**IdealToV2V3**(*v3idlyangle*, *v2ref*, *v3ref*, *vparity*, *name='idl2V'*, ***kwargs*)

Bases: astropy.modeling.core.Model

Performs the transform from Ideal to telescope V2,V3 coordinate system. The two systems have the same origin: V2_REF, V3_REF.

Note: This model has no schema implemented - add schema if needed.

Attributes Summary

inputs	x and y coordinates in the telescope Ideal frame.
outputs	coorinates in the telescope (V2,V3) frame.
param_names	
v2ref	
v3idlyangle	
v3ref	
vparity	

Methods Summary

__call__(xidl, yidl[, model_set_axis, ...])	Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

Continued on next page

Table 430 – continued from previous page

evaluate(xidl, yidl, v3idlyangle, v2ref, ...)	**param xidl, yidl** Coordinates in Ideal System [in arcsec]

Attributes Documentation

inputs = ('xidl', 'yidl')
x and y coordinates in the telescope Ideal frame.

outputs = ('v2', 'v3')
coorinates in the telescope (V2,V3) frame.

param_names = ('v3idlyangle', 'v2ref', 'v3ref', 'vparity')

v2ref

v3idlyangle

v3ref

vparity

Methods Documentation

__call__(*xidl*, *yidl*, *model_set_axis=None*, *with_bounding_box=False*, *fill_value=nan*, *equivalencies=None*)
Evaluate this model using the given input(s) and the parameter values that were specified when the model was instantiated.

static evaluate(*xidl*, *yidl*, *v3idlyangle*, *v2ref*, *v3ref*, *vparity*)

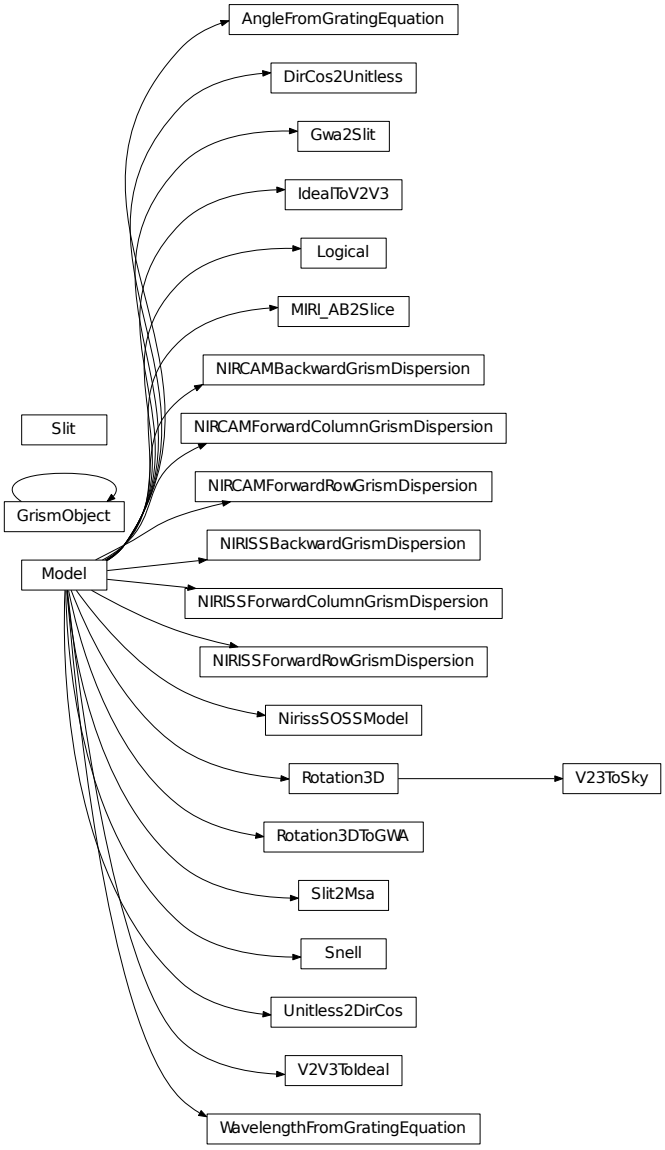
Parameters

- **yidl** (*xidl,*) – Coordinates in Ideal System [in arcsec]
- **v3idlyangle** (*float* (https://docs.python.org/3/library/functions.html#float)) – Angle between Ideal Y-axis and V3 [in deg]
- **v3ref** (*v2ref,*) – Coordinates in V2, V3 [in arcsec]
- **vparity** (*int* (https://docs.python.org/3/library/functions.html#int)) – Parity.

Returns **v2, v3** – Coordinates in the (V2, V3) telescope system [in arcsec].

Return type ndarray-like

Class Inheritance Diagram



12.1.52 TSO Aperture Photometry

Description

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 catalog, a table (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 location of the target is assumed to be given by the CRPIX1 and CRPIX2 FITS keywords (note that these are one-based).

Algorithm

The Astropy affiliated package photutils does the work.

If the input file was not averaged over integrations, and if the file contains an INT_TIMES table, the times shown in the output table will be extracted from column 'int_mid_MJD_UTC' of the INT_TIMES table. Otherwise, the times will be computed from the exposure start time, the group time, and the number of groups in an integration. In either case, the times are Modified Julian Date, time scale UTC.

The output catalog will contain 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.

Reference File

The tso_photometry step uses a TsoPhotModel reference file, reference type TSOPHOT, that supplies values of radius (in pixels) for the target aperture and the inner and outer radii for the background annulus.

CRDS Selection Criteria

TSOPHOT reference files are selected on the basis of INSTRUME, EXP_TYPE, and TSOVISIT. For MIRI exposures, EXP_TYPE should be MIR_IMAGE. For NIRCam exposures, EXP_TYPE should be NRC_TSIMAGE. For both MIRI and NIRCam, TSOVISIT should be True.

Required keywords

These keywords are required to be present in a TsoPhotModel reference file. The first column gives the FITS keyword names (although these reference files are ASDF). The second column gives the model name, which is needed when creating and populating a new reference file.

Keyword	Model Name
AUTHOR	meta.author
DATAMODL	meta.model_type
DATE	meta.data
DESCRIP	meta.description
EXP_TYPE	meta.exposure.type
FILENAME	meta.filename
INSTRUME	meta.instrument.name
PEDIGREE	meta.pedigree
REFTYPE	meta.reftype
TELESCOP	meta.telescope
TSOVISIT	meta.visit.tsovisit
USEAFTER	meta.useafter

TSOPHOT Reference File Format

TSOPHOT reference files are ASDF files. 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'.

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 count rates 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 script itself. The `save_catalog` argument is useful only when the tso_photometry step is run standalone.

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: *jwst.stpipe.Step*

Perform circular aperture photometry on imaging Time Series Observations (TSO).

Parameters **input** (str or CubeModel) – A filename for either a FITS image or and 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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

reference_file_types = ['tsophot']

spec = '\n save_catalog = boolean(default=False) # save exposure-level catalog\n '

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

12.1.53 TweakReg

Description

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 corrected WCS align on sky.

Source detection

Stars are detected in the image using 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.

Alignment

The source catalogs for each input image are compared to each other and linear (affine) coordinate transformations that align these catalogs are derived.

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 using *TPCorr* class and a correcponding object is inserted 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. (Default='False')

- `catalog_format`: A `str` (https://docs.python.org/3/library/stdtypes.html#str) indicating catalog output file format. (Default='ecsv')
- `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)

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=1.0)
- `use2dhist`: A boolean indicating whether to use 2D histogram to find initial offset. (Default=True)
- `separation`: Minimum object separation in arcsec. (Default=0.5)
- `tolerance`: Matching tolerance for `xyxymatch` in arcsec. (Default=1.0)
- `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'`, `'rscale'`, `'general'`}. (Default=''general'')
- `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)

Reference Files

This step does not require any reference files.

Also See:

imalign

A module that provides functions for "aligning" images: specifically, it provides functions for computing corrections to image WCS so that images catalogs "align" to the reference catalog *on the sky*.

Authors Mihai Cara (contact: help@stsci.edu)

jwst.tweakreg.imalign.**align**(*imcat*, *refcat=None*, *enforce_user_order=True*, *expand_refcat=False*, *minobj=None*, *searchrad=1.0*, *use2dhist=True*, *separation=0.5*, *tolerance=1.0*, *xoffset=0.0*, *yoffset=0.0*, *fitgeom='general'*, *nclip=3*, *sigma=3.0*)

Align (groups of) images by adjusting the parameters of their WCS based on fits between matched sources in these images and a reference catalog which may be automatically created from one of the input images.

Parameters

- **imcat** (*list of WCSImageCatalog or* WCSGroupCatalog) – A list of WCSImageCatalog or WCSGroupCatalog objects whose WCS should be adjusted. The difference between WCSImageCatalog and WCSGroupCatalog is that the later is used to find a *single* fit to all sources in all component images *simultaneously*. This fit is later applied to each component image's WCS.

  Warning: This function modifies the WCS of the input images provided through the imcat parameter. On return, each input image WCS will be updated with an "aligned" version.

- **refcat** (RefCatalog, *optional*) – A RefCatalog object that contains a catalog of reference sources as well as (optionally) a valid reference WCS. When refcat is None (https://docs.python.org/3/library/constants.html#None), a reference catalog will be created from one of the input (group of) images.

- **enforce_user_order** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Specifies whether images should be aligned in the order specified in the file input parameter or *align* should optimize the order of alignment by intersection area of the images. Default value (True (https://docs.python.org/3/library/constants.html#True)) will align images in the user specified order, except when some images cannot be aligned in which case *align* will optimize the image alignment order. Alignment order optimization is available *only* when expand_refcat = True (https://docs.python.org/3/library/constants.html#True).

- **expand_refcat** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Specifies whether to add new sources from just matched images to the reference catalog to allow next image to be matched against an expanded reference catalog. By delault, the reference catalog is not being expanded.

- **minobj** (*int* (https://docs.python.org/3/library/functions.html#int), *None* (https://docs.python.org/3/library/constants.html#None), *optional*) – Minimum number of identified objects from each input image to use in matching objects from other images. If the default None (https://docs.python.org/3/library/constants.html#None) value is used then *align* will automatically determnine the minimum number of sources from the value of the fitgeom parameter.

- **searchrad** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The search radius for a match.

- **use2dhist** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Use 2D histogram to find initial offset?

- **separation** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The minimum separation for sources in the input and reference catalogs in order to be considered to be disctinct sources. Objects closer together than 'separation' pixels are removed from the input and reference coordinate lists prior to matching. This parameter gets passed directly to xyxymatch() for use in matching the object lists from each image with the reference image's object list.

- **tolerance** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The matching tolerance in pixels after applying an initial solution derived from the 'triangles' algorithm. This parameter gets passed directly to `xyxymatch()` for use in matching the object lists from each image with the reference image's object list.
- **xoffset** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Initial estimate for the offset in X between the images and the reference frame. This offset will be used for all input images provided. This parameter is ignored when `use2dhist` is `True` (https://docs.python.org/3/library/constants.html#True).
- **yoffset** (*float* (https://docs.python.org/3/library/functions.html#float) *(Default = 0.0)*) – Initial estimate for the offset in Y between the images and the reference frame. This offset will be used for all input images provided. This parameter is ignored when `use2dhist` is `True` (https://docs.python.org/3/library/constants.html#True).
- **fitgeom** (*{'shift', 'rscale', 'general'}, optional*) – The fitting geometry to be used in fitting the matched object lists. This parameter is used in fitting the offsets, rotations and/or scale changes from the matched object lists. The 'general' fit geometry allows for independent scale and rotation for each axis.
- **nclip** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – Number (a non-negative integer) of clipping iterations in fit.
- **sigma** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Clipping limit in sigma units.

`jwst.tweakreg.imalign.`**overlap_matrix**(*images*)

Compute overlap matrix: non-diagonal elements (i,j) of this matrix are absolute value of the area of overlap on the sky between i-th input image and j-th input image.

Note: The diagonal of the returned overlap matrix is set to `0.0`, i.e., this function does not compute the area of the footprint of a single image on the sky.

Parameters **images** (*list of WCSImageCatalog, WCSGroupCatalog, or RefCatalog*) – A list of catalogs that implement `intersection_area()` method.

Returns **m** – A `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.nda of shape `NxN` where `N` is equal to the number of input images. Each non-diagonal element (i,j) of this matrix is the absolute value of the area of overlap on the sky between i-th input image and j-th input image. Diagonal elements are set to `0.0`.

Return type numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray)

`jwst.tweakreg.imalign.`**max_overlap_pair**(*images*, *enforce_user_order*)

Return a pair of images with the largest overlap.

Warning: Returned pair of images is "poped" from input `images` list and therefore on return `images` will contain a smaller number of elements.

Parameters

- **images** (*list of WCSImageCatalog, WCSGroupCatalog, or RefCatalog*) – A list of catalogs that implement `intersection_area()` method.
- **enforce_user_order** (*bool* (https://docs.python.org/3/library/functions.html#bool)) – When `enforce_user_order` is `True` (https://docs.python.org/3/library/constants.html#True),

a pair of images will be returned **in the same order** as they were arranged in the `images` input list. That is, image overlaps will be ignored.

Returns Returns a tuple of two images - elements of input `images` list. When `enforce_user_order` is `True` (https://docs.python.org/3/library/constants.html#True), images are returned in the order in which they appear in the input `images` list. When the number of input images is smaller than two, `im1` and `im2` may be `None` (https://docs.python.org/3/library/constants.html#None).

Return type (im1, im2)

`jwst.tweakreg.imalign.`**`max_overlap_image`**(*refimage*, *images*, *enforce_user_order*)

Return the image from the input `images` list that has the largest overlap with the `refimage` image.

Warning: Returned image of images is "poped" from input `images` list and therefore on return `images` will contain a smaller number of elements.

Parameters

- **`images`** (*`list of WCSImageCatalog, or WCSGroupCatalog`*) – A list of catalogs that implement `intersection_area()` method.
- **`enforce_user_order`** (*`bool`* (https://docs.python.org/3/library/functions.html#bool)) – When `enforce_user_order` is `True` (https://docs.python.org/3/library/constants.html#True), returned image is the first image from the `images` input list regardless ofimage overlaps.

Returns **image** – Returns an element of input `images` list. When input list is empty - `None` (https://docs.python.org/3/library/constants.html#None) is returned.

Return type *WCSImageCatalog*, *WCSGroupCatalog*, or None (https://docs.python.org/3/library/constants.html#None)

wcsimage

This module provides support for working with image footprints on the sky, source catalogs, and setting and manipulating tangent-plane corrections of image WCS.

Authors Mihai Cara (contact: help@stsci.edu)

`jwst.tweakreg.wcsimage.`**`convex_hull`**(*x*, *y*, *wcs=None*)

Computes the convex hull of a set of 2D points.

Implements Andrew's monotone chain algorithm (http://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hu
The algorithm has O(n log n) complexity.

Credit: http://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain

Parameters **`points`** (*`list of tuples`*) – An iterable sequence of (x, y) pairs representing the points.

Returns **Output** – A list of vertices of the convex hull in counter-clockwise order, starting from the vertex with the lexicographically smallest coordinates.

Return type list (https://docs.python.org/3/library/stdtypes.html#list)

`class` `jwst.tweakreg.wcsimage.`**`ImageWCS`**(*wcs*, *v2_ref*, *v3_ref*, *roll_ref*, *ra_ref*, *dec_ref*)

A class for holding JWST GWCS information and for managing tangent-plane corrections.

Parameters

- **wcs** (*GWCS*) – A `GWCS` object.
- **v2ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – V2 position of the reference point in degrees.
- **v3ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – V3 position of the reference point in degrees.
- **roll** (*float* (https://docs.python.org/3/library/functions.html#float)) – Roll angle in degrees.
- **ra_ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – RA of the reference point in degrees.
- **dec_ref** (*float* (https://docs.python.org/3/library/functions.html#float)) – DEC of the reference point in degrees.

copy()
Returns a deep copy of this *ImageWCS* object.

det_to_tanp(*x*, *y*)
Convert detector (pixel) coordinates to tangent plane coordinates.

det_to_world(*x*, *y*)
Convert pixel coordinates to sky coordinates using full (i.e., including distortions) transformations.

original_wcs
Get original GWCS object.

ref_angles
Return a `wcsinfo`-like dictionary of main WCS parameters.

set_correction(*matrix=[[1, 0], [0, 1]], shift=[0, 0]*)
Sets a tangent-plane correction of the GWCS object according to the provided liniar parameters.

Parameters

- **matrix** (*list* (https://docs.python.org/3/library/stdtypes.html#list), *numpy.ndarray* (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray)) – A `2x2` array or list of lists coefficients representing scale, rotation, and/or skew transformations.
- **shift** (*list* (https://docs.python.org/3/library/stdtypes.html#list), *numpy.ndarray* (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray)) – A list of two coordinate shifts to be applied to coordinates *before* `matrix` transformations are applied.

tanp_to_det(*x*, *y*)
Convert tangent plane coordinates to detector (pixel) coordinates.

tanp_to_world(*x*, *y*)
Convert tangent plane coordinates to world coordinates.

wcs
Get current GWCS object.

world_to_det(*ra*, *dec*)
Convert sky coordinates to image's pixel coordinates using full (i.e., including distortions) transformations.

world_to_tanp(*ra*, *dec*)
Convert tangent plane coordinates to detector (pixel) coordinates.

class jwst.tweakreg.wcsimage.**RefCatalog**(*catalog*, *name=None*)

An object that holds a reference catalog and provides tools for coordinate convertions using reference WCS as well as catalog manipulation and expansion.

Parameters

- **catalog** (*astropy.table.Table*) – Reference catalog.

  ..note:: Reference catalogs (Table) *must* contain *both* 'RA' and 'DEC' columns.

- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *None* (https://docs.python.org/3/library/constants.html#None), *optional*) – Name of the reference catalog.

calc_bounding_polygon()

Calculate bounding polygon of the sources in the catalog.

calc_tanp_xy(*tanplane_wcs*)

Compute x- and y-positions of the sources from the reference catalog in the tangent plane provided by the tanplane_wcs. This creates the following columns in the catalog's table: 'xtanp' and 'ytanp'.

Parameters **tanplane_wcs** (ImageWCS) – A *ImageWCS* object that will provide transformations to the tangent plane to which sources of this catalog a should be "projected".

catalog

Get/set image's catalog.

expand_catalog(*catalog*)

Expand current reference catalog with sources from another catalog.

Parameters **catalog** (*astropy.table.Table*) – A catalog of new sources to be added to the existing reference catalog. *catalog* *must* contain *both* 'RA' and 'DEC' columns.

intersection(*wcsim*)

Compute intersection of this *WCSImageCatalog* object and another *WCSImageCatalog*, *WCSGroupCatalog*, *RefCatalog*, or SphericalPolygon object.

Parameters **wcsim** (WCSImageCatalog, WCSGroupCatalog, RefCatalog, *SphericalPolygon*) – Another object that should be intersected with this *WCSImageCatalog*.

Returns **polygon** – A SphericalPolygon that is the intersection of this *WCSImageCatalog* and wcsim.

Return type SphericalPolygon

intersection_area(*wcsim*)

Calculate the area of the intersection polygon.

name

Get/set *WCSImageCatalog* object's name.

poly_area

Area of the bounding polygon (in srad).

polygon

Get image's (or catalog's) bounding spherical polygon.

class jwst.tweakreg.wcsimage.**WCSImageCatalog**(*shape*, *wcs*, *ref_angles*, *catalog*, *name=None*, *meta={}*)

A class that holds information pertinent to an image WCS and a source catalog of the sources found in that image.

Parameters

- **shape** (*tuple* (https://docs.python.org/3/library/stdtypes.html#tuple)) – A tuple of two integer values indicating the size of the image along each axis. Must follow the same convention as the shape of a `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) objects. Specifically, first size should be indicate the number of rows in the image and second size should indicate the number of columns in the image.
- **wcs** (*gwcs.WCS*) – WCS associated with the image and the catalog.
- **ref_angles** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – A Python dictionary providing essential WCS reference angles. This parameter must contain at least the following keys: `ra_ref`, `dec_ref`, `v2_ref`, `v3_ref`, and `roll_ref`.
- **catalog** (*astropy.table.Table*) – Source catalog associated with an image. Must contain 'x' and 'y' columns which indicate source coordinates (in pixels) in the associated image.
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *None* (https://docs.python.org/3/library/constants.html#None), *optional*) – Image catalog's name.
- **meta** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict), *optional*) – Additional information about image, catalog, and/or WCS to be stored (for convenience) within *WCSImageCatalog* object.

bb_radec
Get a 2xN `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) of RA and DEC of the vertices of the bounding polygon.

calc_bounding_polygon()
Calculate bounding polygon of the image or of the sources in the catalog (if catalog was set).

catalog
Get/set image's catalog.

det_to_tanp(*x*, *y*)
Convert detector (pixel) coordinates to tangent plane coordinates.

det_to_world(*x*, *y*)
Convert pixel coordinates to sky coordinates using full (i.e., including distortions) transformations.

imshape
Get/set image's shape. This must be a tuple of two dimensions following the same convention as the shape of `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray).

imwcs
Get *ImageWCS* WCS.

intersection(*wcsim*)
Compute intersection of this *WCSImageCatalog* object and another *WCSImageCatalog*, *WCSGroupCatalog*, or `SphericalPolygon` object.

Parameters **wcsim** (`WCSImageCatalog`, `WCSGroupCatalog`, *SphericalPolygon*) – Another object that should be intersected with this *WCSImageCatalog*.

Returns **polygon** – A `SphericalPolygon` that is the intersection of this *WCSImageCatalog* and `wcsim`.

Return type SphericalPolygon

intersection_area (*wcsim*)
Calculate the area of the intersection polygon.

name
Get/set *WCSImageCatalog* object's name.

polygon
Get image's (or catalog's) bounding spherical polygon.

ref_angles
Get `wcsinfo`.

set_wcs (*wcs*, *ref_angles*)
Set `gwcs.WCS` and the associated `wcsinfo`.

Note: Setting the WCS triggers automatic bounding polygon recalculation.

Parameters

- **wcs** (*gwcs.WCS*) – WCS object.
- **ref_angles** (*dict* (https://docs.python.org/3/library/stdtypes.html#dict)) – A Python dictionary providing essential WCS reference angles. This parameter must contain at least the following keys: `ra_ref`, `dec_ref`, `v2_ref`, `v3_ref`, and `roll_ref`.

tanp_to_det (*x*, *y*)
Convert tangent plane coordinates to detector (pixel) coordinates.

tanp_to_world (*x*, *y*)
Convert tangent plane coordinates to world coordinates.

wcs
Get `gwcs.WCS`.

world_to_det (*ra*, *dec*)
Convert sky coordinates to image's pixel coordinates using full (i.e., including distortions) transformations.

world_to_tanp (*ra*, *dec*)
Convert tangent plane coordinates to detector (pixel) coordinates.

class jwst.tweakreg.wcsimage.**WCSGroupCatalog** (*images*, *name=None*)
A class that holds together *WCSImageCatalog* image catalog objects whose relative positions are fixed and whose source catalogs should be fitted together to a reference catalog.

Parameters

- **images** (*list of WCSImageCatalog*) – A list of *WCSImageCatalog* image catalogs.
- **name** (*str* (https://docs.python.org/3/library/stdtypes.html#str), *None* (https://docs.python.org/3/library/constants.html#None), *optional*) – Name of the group.

align_to_ref (*refcat*, *minobj=15*, *searchrad=1.0*, *separation=0.5*, *use2dhist=True*, *xoffset=0.0*, *yoffset=0.0*, *tolerance=1.0*, *fitgeom='rscale'*, *nclip=3*, *sigma=3.0*)
Matches sources from the image catalog to the sources in the reference catalog, finds the affine transformation between matched sources, and adjusts images' WCS according to this fit.

Parameters

- **refcat** (RefCatalog) – A *RefCatalog* object that contains a catalog of reference sources as well as a valid reference WCS.
- **minobj** (*int* (https://docs.python.org/3/library/functions.html#int), *None* (https://docs.python.org/3/library/constants.html#None), *optional*) – Minimum number of identified objects from each input image to use in matching objects from other images. If the default None (https://docs.python.org/3/library/constants.html#None) value is used then align will automatically determnine the minimum number of sources from the value of the fitgeom parameter.
- **searchrad** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The search radius for a match.
- **separation** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The minimum separation for sources in the input and reference catalogs in order to be considered to be disctinct sources. Objects closer together than 'separation' pixels are removed from the input and reference coordinate lists prior to matching. This parameter gets passed directly to xyxymatch() for use in matching the object lists from each image with the reference image's object list.
- **use2dhist** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Use 2D histogram to find initial offset?
- **xoffset** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Initial estimate for the offset in X between the images and the reference frame. This offset will be used for all input images provided. This parameter is ignored when use2dhist is True (https://docs.python.org/3/library/constants.html#True).
- **yoffset** (*float* (https://docs.python.org/3/library/functions.html#float) *(Default = 0.0)*) – Initial estimate for the offset in Y between the images and the reference frame. This offset will be used for all input images provided. This parameter is ignored when use2dhist is True (https://docs.python.org/3/library/constants.html#True).
- **tolerance** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The matching tolerance in pixels after applying an initial solution derived from the 'triangles' algorithm. This parameter gets passed directly to xyxymatch() for use in matching the object lists from each image with the reference image's object list.
- **fitgeom** (*{'shift', 'rscale', 'general'}, optional*) – The fitting geometry to be used in fitting the matched object lists. This parameter is used in fitting the offsets, rotations and/or scale changes from the matched object lists. The 'general' fit geometry allows for independent scale and rotation for each axis.
- **nclip** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – Number (a non-negative integer) of clipping iterations in fit.
- **sigma** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Clipping limit in sigma units.

apply_affine_to_wcs(*tanplane_wcs*, *matrix*, *shift*)
Applies a general affine transformation to the WCS.

calc_tanp_xy(*tanplane_wcs*)
Compute x- and y-positions of the sources from the image catalog in the tangent plane. This creates the following columns in the catalog's table: 'xtanp' and 'ytanp'.

Parameters **tanplane_wcs** (ImageWCS) – A *ImageWCS* object that will provide transformations to the tangent plane to which sources of this catalog a should be "projected".

catalog
Get/set image's catalog.

create_group_catalog()
Combine member's image catalogs into a single group's catalog.

Returns **group_catalog** – Combined group catalog.

Return type astropy.table.Table

fit2ref(*refcat*, *tanplane_wcs*, *fitgeom='general'*, *nclip=3*, *sigma=3.0*)
Perform linear fit of this group's combined catalog to the reference catalog.

Parameters

- **refcat** (RefCatalog) – A *RefCatalog* object that contains a catalog of reference sources.
- **tanplane_wcs** (ImageWCS) – A *ImageWCS* object that will provide transformations to the tangent plane to which sources of this catalog a should be "projected".
- **fitgeom** (*{'shift', 'rscale', 'general'}, optional*) – The fitting geometry to be used in fitting the matched object lists. This parameter is used in fitting the offsets, rotations and/or scale changes from the matched object lists. The 'general' fit geometry allows for independent scale and rotation for each axis.
- **nclip** (*int* (https://docs.python.org/3/library/functions.html#int), *optional*) – Number (a non-negative integer) of clipping iterations in fit.
- **sigma** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Clipping limit in sigma units.

get_matched_cat()
Retrieve only those sources from the catalog that **have been** matched to the sources in the reference catalog.

get_unmatched_cat()
Retrieve only those sources from the catalog that have **not** been matched to the sources in the reference catalog.

intersection(*wcsim*)
Compute intersection of this *WCSGroupCatalog* object and another *WCSImageCatalog*, *WCSGroupCatalog*, or SphericalPolygon object.

Parameters **wcsim** (WCSImageCatalog, WCSGroupCatalog, *SphericalPolygon*) – Another object that should be intersected with this *WCSGroupCatalog*.

Returns **polygon** – A SphericalPolygon that is the intersection of this *WCSGroupCatalog* and wcsim.

Return type SphericalPolygon

intersection_area(*wcsim*)
Calculate the area of the intersection polygon.

match2ref(*refcat*, *minobj=15*, *searchrad=1.0*, *separation=0.5*, *use2dhist=True*, *xoffset=0.0*, *yoffset=0.0*, *tolerance=1.0*)
Uses xyxymatch to cross-match sources between this catalog and a reference catalog.

Parameters

- **refcat** (RefCatalog) – A *RefCatalog* object that contains a catalog of reference sources as well as a valid reference WCS.
- **minobj** (*int* (https://docs.python.org/3/library/functions.html#int), *None* (https://docs.python.org/3/library/constants.html#None), *optional*) – Minimum

number of identified objects from each input image to use in matching objects from other images. If the default `None` (https://docs.python.org/3/library/constants.html#None) value is used then `align` will automatically determnine the minimum number of sources from the value of the `fitgeom` parameter.

- **searchrad** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The search radius for a match.
- **separation** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The minimum separation for sources in the input and reference catalogs in order to be considered to be disctinct sources. Objects closer together than 'separation' pixels are removed from the input and reference coordinate lists prior to matching. This parameter gets passed directly to `xyxymatch()` for use in matching the object lists from each image with the reference image's object list.
- **use2dhist** (*bool* (https://docs.python.org/3/library/functions.html#bool), *optional*) – Use 2D histogram to find initial offset?
- **xoffset** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Initial estimate for the offset in X between the images and the reference frame. This offset will be used for all input images provided. This parameter is ignored when `use2dhist` is `True` (https://docs.python.org/3/library/constants.html#True).
- **yoffset** (*float* (https://docs.python.org/3/library/functions.html#float) *(Default = 0.0)*) – Initial estimate for the offset in Y between the images and the reference frame. This offset will be used for all input images provided. This parameter is ignored when `use2dhist` is `True` (https://docs.python.org/3/library/constants.html#True).
- **tolerance** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The matching tolerance in pixels after applying an initial solution derived from the 'triangles' algorithm. This parameter gets passed directly to `xyxymatch()` for use in matching the object lists from each image with the reference image's object list.

name
Get/set *WCSImageCatalog* object's name.

polygon
Get image's (or catalog's) bounding spherical polygon.

recalc_catalog_radec()
Recalculate RA and DEC of the sources in the catalog.

update_bounding_polygon()
Recompute bounding polygons of the member images.

linearfit

A module that provides algorithms for performing linear fits between sets of 2D points.

Authors Mihai Cara, Warren Hack (contact: help@stsci.edu)

jwst.tweakreg.linearfit.**iter_linear_fit**(*xy*, *uv*, *xyindx=None*, *uvindx=None*, *xyorig=None*, *uvorig=None*, *fitgeom='general'*, *nclip=3*, *sigma=3.0*, *center=None*)
Compute iteratively using sigma-clipping linear transformation parameters that fit `xy` sources to `uv` sources.

jwst.tweakreg.linearfit.**build_fit_matrix**(*rot*, *scale=1*)
Create an affine transformation matrix (2x2) from the provided rotation and scale transformations.

Parameters

- **rot** (*tuple* (https://docs.python.org/3/library/stdtypes.html#tuple), *float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Rotation angle in degrees. Two values (one for each axis) can be provided as a tuple.
- **scale** (*tuple* (https://docs.python.org/3/library/stdtypes.html#tuple), *float* (https://docs.python.org/3/library/functions.html#float), *optional*) – Scale of the liniar transformation. Two values (one for each axis) can be provided as a tuple.

Returns **matrix** – A 2x2 numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#nu containing coefficients of a liniear transformation.

Return type numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray)

matchutils

A module that provides algorithms for initial estimation of shifts based on 2D histograms.

jwst.tweakreg.matchutils.**build_xy_zeropoint**(*imgxy*, *refxy*, *searchrad=3.0*)
Create a matrix which contains the delta between each XY position and each UV position.

jwst.tweakreg.matchutils.**center_of_mass**(*img*, *labels=None*, *index=None*)
Calculate the center of mass of the values of an array at labels.

Parameters **img** (*ndarray*) – Data from which to calculate center-of-mass.

Returns **centerofmass** – Coordinates of centers-of-masses.

Return type tuple (https://docs.python.org/3/library/stdtypes.html#tuple), or list of tuples

Examples

```
>>> from jwst.tweakreg import matchutils
>>> a = np.array(([0,0,0,0],
                  [0,1,1,0],
                  [0,1,1,0],
                  [0,1,1,0]))
>>> matchutils.center_of_mass(a)
(2.0, 1.5)
```

jwst.tweakreg.matchutils.**find_xy_peak**(*img*, *center=None*, *sigma=3.0*)
Find the center of the peak of offsets

tweakreg_catalog

The tweakreg_catalog module provides functions for generating catalogs of sources from images.

jwst.tweakreg.tweakreg_catalog.**make_tweakreg_catalog**(*model*, *kernel_fwhm*, *snr_threshold*, *sharplo=0.2*, *sharphi=1.0*, *roundlo=-1.0*, *roundhi=1.0*)
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 roundess for object detection.
- **roundhi** (*float* (https://docs.python.org/3/library/functions.html#float), *optional*) – The upper bound on roundess for object detection.

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 pipeline step for image alignment.

Authors Mihai Cara

class jwst.tweakreg.tweakreg_step.**TweakRegStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

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.
- ****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.

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.

reference_file_types = []

spec = "\n # Source finding parameters:\n save_catalogs = boolean(default=False) # Wri

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: *jwst.stpipe.Step*

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.
- ****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

reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

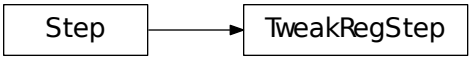
reference_file_types = []

spec = "\n # Source finding parameters:\n save_catalogs = boolean(default=False) # Wri

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



12.1.54 WFS Combine

Description

This step takes as input a set of 2 dithered wave front sensing images. The names of these images and the name of the output are given in an association table. The table can contain a list of several combined products to be created (each from a separate pair of input files). Each pair of input images is 'combined' by:

1. WCS information is read from both images, from which the difference in pointings (in pixels) is calculated
2. Image #2 is aligned in the frame of image #1 using this WCS information in the input headers
3. For each pixel in the overlapped region, construct a 'combined' SCI image using:
   a) the pixel from image #1 if that pixel has a good DQ value, else
   b) the pixel from image #2 if that pixel has a good DQ value, else
   c) a default value (0).
4. For each pixel in the overlapped region, construct a 'combined' Data Quality image using:
   a) the DQ pixel from image #1 if that pixel has a good DQ value, else
   b) the DQ pixel from image #2 if that pixel has a good DQ value, else
   c) a default 'BAD_WFS' value added to the corresponding value in image #1.
5. For each pixel in the overlapped region, construct a 'combined' Error image using:
   a) the ERR pixel from image #1 if that pixel has a good DQ value, else
   b) the ERR pixel from image #2 if that pixel has a good DQ value, else
   c) a default value(0).

If the option to refine the estimate of the offsets is chosen (this is not the default) step #2 above becomes:

2. a) Interpolate over missing data (based on the corresponding DQ array) in both images
   b) Align these interpolated images to a common frame using the WCS information in the input headers

c) Compare the 2 nominally aligned, interpolated images by varying the offsets to have values in the neighborhood of the nominal offsets to determine the best match.

d) Align the original (pre-interpolated) image #2 in the frame of image #1 using this refined estimate of the offsets

Upon successful completion of this step, the status keyword S_WFSCOM will be set to COMPLETE.

Reference File

The wave front sensing combination step does not use any reference files.

WAVE FRONT SENSING COMBINATION

This module combines 2 dithered wave front sensing images

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: *jwst.stpipe.Step*

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.
- ****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

spec

Methods Summary

process(input_table)	This is where real work happens.

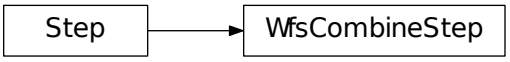
Attributes Documentation

spec = '\n do_refine = boolean(default=False)\n '

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



12.1.55 White Light Curve Generation

Description

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 a target. This is to be applied to the `_x1dints` product in a spectroscopic Time-Series Observation (TSO), as part of the `calwebb_tso3` pipeline.

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.

Output product

The output product is a table of time vs. integrated flux, stored in the form of a ASCII ECSV (Extended Comma-Separated Value) file. The product type suffix is `_whtlt`.

Reference File

The white_light step does not use any reference files.

Step Arguments

The white light curve generation step has no step-specific arguments.

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: *jwst.stpipe.Step*

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.
- ****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

spec	

Methods Summary

process(input)	This is where real work happens.

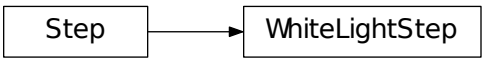
Attributes Documentation

spec = "\n output_ext = string(default='.ecsv') # Default type of output\n suffix = st

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



12.1.56 Weighted Image Intensity Matching

`wiimatch` is a package that provides core computational algorithms for optimal "matching" of weighted N-dimensional image intensity data using (multivariate) polynomials.

LSQ Image Intensity Matching

A module that provides main API for optimal (LSQ) "matching" of weighted N-dimensional image intensity data using (multivariate) polynomials.

Author Mihai Cara (contact: help@stsci.edu)

`jwst.wiimatch.match.`**match_lsq**(*images*, *masks=None*, *sigmas=None*, *degree=0*, *center=None*, *image2world=None*, *center_cs='image'*, *ext_return=False*, *solver='RLU'*)

Compute coefficients of (multivariate) polynomials that once subtracted from input images would provide image intensity matching in the least squares sense.

images [list of numpy.ndarray] A list of 1D, 2D, etc. `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) data array whose "intensities" must be "matched". All arrays must have identical shapes.

masks [list of numpy.ndarray, None] A list of `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) arrays of same length as `images`. Non-zero mask elements indicate valid data in the corresponding `images` array. Mask arrays must have identical shape to that of the arrays in

input images. Default value of None (https://docs.python.org/3/library/constants.html#None) indicates that all pixels in input images are valid.

sigmas [list of numpy.ndarray, None] A list of numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) data array of same length as images representing the uncertainties of the data in the corresponding array in images. Uncertainty arrays must have identical shape to that of the arrays in input images. The default value of None (https://docs.python.org/3/library/constants.html#None) indicates that all pixels will be assigned equal weights.

degree [iterable, int] A list of polynomial degrees for each dimension of data arrays in images. The length of the input list must match the dimensionality of the input images. When a single integer number is provided, it is assumed that the polynomial degree in each dimension is equal to that integer.

center [iterable, None, optional] An iterable of length equal to the number of dimensions in image_shape that indicates the center of the coordinate system in **image** coordinates when center_cs is 'image' otherwise center is assumed to be in **world** coordinates (when center_cs is 'world'). When center is None (https://docs.python.org/3/library/constants.html#None) then center is set to the middle of the "image" as center[i]=image_shape[i]//2. If image2world is not None (https://docs.python.org/3/library/constants.html#None) and center_cs is 'image', then supplied center will be converted to world coordinates.

image2world [function, None, optional] Image-to-world coordinates transformation function. This function must be of the form f(x, y,z,...) and accept a number of arguments numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) arguments equal to the dimensionality of images.

center_cs [{'image', 'world'}, optional] Indicates whether center is in image coordinates or in world coordinates. This parameter is ignored when center is set to None (https://docs.python.org/3/library/constants.html#None): it is assumed to be False (https://docs.python.org/3/library/constants.html#False). center_cs *cannot be* 'world' when image2world is None (https://docs.python.org/3/library/constants.html#None) unless center is None (https://docs.python.org/3/library/constants.html#None).

ext_return [bool, optional] Indicates whether this function should return additional values besides optimal polynomial coefficients (see bkg_poly_coeff return value below) that match image intensities in the LSQ sense. See **Returns** section for more details.

solver [{'RLU', 'PINV'}, optional] Specifies method for solving the system of equations.

bkg_poly_coeff [numpy.ndarray] When nimages is None (https://docs.python.org/3/library/constants.html#None), this function returns a 1D numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.htm that holds the solution (polynomial coefficients) to the system.

When nimages is **not** None (https://docs.python.org/3/library/constants.html#None), this function returns a 2D numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#n that holds the solution (polynomial coefficients) to the system. The solution is grouped by image.

a [numpy.ndarray] A 2D numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#nu that holds the coefficients of the linear system of equations. This value is returned only when ext_return is True (https://docs.python.org/3/library/constants.html#True).

b [numpy.ndarray] A 1D numpy.ndarray (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#nu that holds the free terms of the linear system of equations. This value is returned only when ext_return is True (https://docs.python.org/3/library/constants.html#True).

coord_arrays [list] A list of `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html) coordinate arrays each of `image_shape` shape. This value is returned only when `ext_return` is `True` (https://docs.python.org/3/library/constants.html#True).

eff_center [tuple] A tuple of coordinates of the effective center as used in generating coordinate arrays. This value is returned only when `ext_return` is `True` (https://docs.python.org/3/library/constants.html#True).

coord_system [{'image', 'world'}] Coordinate system of the coordinate arrays and returned `center` value. This value is returned only when `ext_return` is `True` (https://docs.python.org/3/library/constants.html#True).

`match_lsq()` 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 intensities of 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 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:

$$P_n(k_1, k_2, ...) = \sum_{d_1=0,d_2=0,...}^{D_1,D_2,...} c^n_{d_1,d_2,...} \cdot k_1^{d_1} \cdot k_2^{d_2} \cdot ....$$

Coefficients $c^n_{d_1,d_2,...}$ are arranged in the vector c in the following order:

$$(c^1_{0,0,...}, c^1_{1,0,...}, \ldots, c^2_{0,0,...}, c^2_{1,0,...}, \ldots).$$

`match_lsq()` returns coefficients of the polynomials that minimize L.

```
>>> import wiimatch
>>> import numpy as np
>>> im1 = np.zeros((5, 5, 4), dtype=np.float)
>>> cbg = 1.32 * np.ones_like(im1)
>>> ind = np.indices(im1.shape, dtype=np.float)
>>> im3 = cbg + 0.15 * ind[0] + 0.62 * ind[1] + 0.74 * ind[2]
>>> mask = np.ones_like(im1, dtype=np.int8)
>>> sigma = np.ones_like(im1, dtype=np.float)
>>> wiimatch.match.match_lsq([im1, im3], [mask, mask], [sigma, sigma],
... degree=(1, 1, 1), center=(0, 0, 0))
array([[ -6.60000000e-01,  -7.50000000e-02,  -3.10000000e-01,
          3.33066907e-15,  -3.70000000e-01,   5.44009282e-15,
          7.88258347e-15,  -2.33146835e-15],
       [  6.60000000e-01,   7.50000000e-02,   3.10000000e-01,
         -4.44089210e-15,   3.70000000e-01,  -4.21884749e-15,
         -7.43849426e-15,   1.77635684e-15]])
```

LSQ Equation Construction and Solving

A module that provides core algorithm for optimal matching of backgrounds of N-dimensional images using (multivariate) polynomials.

Author Mihai Cara (contact: help@stsci.edu)

jwst.wiimatch.lsq_optimizer.**build_lsq_eqs**(*images*, *masks*, *sigmas*, *degree*, *center=None*, *image2world=None*, *center_cs='image'*)

Build system of linear equations whose solution would provide image intensity matching in the least squares sense.

images [list of numpy.ndarray] A list of 1D, 2D, etc. `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) data array whose "intensities" must be "matched". All arrays must have identical shapes.

masks [list of numpy.ndarray] A list of `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) arrays of same length as `images`. Non-zero mask elements indicate valid data in the corresponding `images` array. Mask arrays must have identical shape to that of the arrays in input `images`.

sigmas [list of numpy.ndarray] A list of `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) data array of same length as `images` representing the uncertainties of the data in the corresponding array in `images`. Uncertainty arrays must have identical shape to that of the arrays in input `images`.

degree [iterable] A list of polynomial degrees for each dimension of data arrays in `images`. The length of the input list must match the dimensionality of the input images.

center [iterable, None, optional] An iterable of length equal to the number of dimensions of images in `images` parameter that indicates the center of the coordinate system in **image** coordinates when `center_cs` is `'image'` otherwise center is assumed to be in **world** coordinates (when `center_cs` is `'world'`). When `center` is `None` (https://docs.python.org/3/library/constants.html#None) then `center` is set to the middle of the "image" as `center[i]=image.shape[i]//2`. If `image2world` is not `None` (https://docs.python.org/3/library/constants.html#None) and `center_cs` is `'image'`, then supplied center will be converted to world coordinates.

image2world [function, None, optional] Image-to-world coordinates transformation function. This function must be of the form `f(x, y, z, ...)` and accept a number of arguments `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) arguments equal to the dimensionality of images.

center_cs [{'image', 'world'}, optional] Indicates whether `center` is in image coordinates or in world coordinates. This parameter is ignored when `center` is set to `None` (https://docs.python.org/3/library/constants.html#None): it is assumed to be `False` (https://docs.python.org/3/library/constants.html#False). `center_cs` *cannot be* `'world'` when `image2world` is `None` (https://docs.python.org/3/library/constants.html#None) unless `center` is `None` (https://docs.python.org/3/library/constants.html#None).

a [numpy.ndarray] A 2D `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#nu that holds the coefficients of the linear system of equations.

b [numpy.ndarray] A 1D `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#nu that holds the free terms of the linear system of equations.

coord_arrays [list] A list of `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html coordinate arrays each of `images[0].shape` shape.

eff_center [tuple] A tuple of coordinates of the effective center as used in generating coordinate arrays.

coord_system [{'image', 'world'}] Coordinate system of the coordinate arrays and returned `center` value.

`build_lsq_eqs()` 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 intensities of input images) such that the following sum is minimized:

$$L = \sum_{n,m=1,n\neq m}^{N} \sum_{k} \frac{\left[I_n(k) - I_m(k) - P_n(k) + P_m(k)\right]^2}{\sigma_n^2(k) + \sigma_m^2(k)}.$$

In the above equation, index $k = (k_1, k_2, ...)$ labels a position in 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:

$$P_n(k_1, k_2, ...) = \sum_{d_1=0,d_2=0,...}^{D_1,D_2,...} c^n_{d_1,d_2,...} \cdot k_1^{d_1} \cdot k_2^{d_2} \cdot ....$$

Coefficients $c^n_{d_1,d_2,...}$ are arranged in the vector c in the following order:

$$(c^1_{0,0,...}, c^1_{1,0,...}, \ldots, c^2_{0,0,...}, c^2_{1,0,...}, \ldots).$$

```
>>> import wiimatch
>>> import numpy as np
>>> im1 = np.zeros((5, 5, 4), dtype=np.float)
>>> cbg = 1.32 * np.ones_like(im1)
>>> ind = np.indices(im1.shape, dtype=np.float)
>>> im3 = cbg + 0.15 * ind[0] + 0.62 * ind[1] + 0.74 * ind[2]
>>> mask = np.ones_like(im1, dtype=np.int8)
>>> sigma = np.ones_like(im1, dtype=np.float)
>>> a, b, ca, ef, cs = wiimatch.lsq_optimizer.build_lsq_eqs([im1, im3],
... [mask, mask], [sigma, sigma], degree=(1,1,1), center=(0,0,0))
>>> print(a)
[[   50.   100.   100.   200.    75.   150.   150.   300.   -50.  -100.
   -100.  -200.   -75.  -150.  -150.  -300.]
   [  100.   300.   200.   600.   150.   450.   300.   900.  -100.  -300.
   -200.  -600.  -150.  -450.  -300.  -900.]
   [  100.   200.   300.   600.   150.   300.   450.   900.  -100.  -200.
   -300.  -600.  -150.  -300.  -450.  -900.]
   [  200.   600.   600.  1800.   300.   900.   900.  2700.  -200.  -600.
   -600. -1800.  -300.  -900.  -900. -2700.]
   [   75.   150.   150.   300.   175.   350.   350.   700.   -75.  -150.
   -150.  -300.  -175.  -350.  -350.  -700.]
   [  150.   450.   300.   900.   350.  1050.   700.  2100.  -150.  -450.
   -300.  -900.  -350. -1050.  -700. -2100.]
   [  150.   300.   450.   900.   350.   700.  1050.  2100.  -150.  -300.
   -450.  -900.  -350.  -700. -1050. -2100.]
   [  300.   900.   900.  2700.   700.  2100.  2100.  6300.  -300.  -900.
   -900. -2700.  -700. -2100. -2100. -6300.]
   [  -50.  -100.  -100.  -200.   -75.  -150.  -150.  -300.    50.   100.
   100.   200.    75.   150.   150.   300.]
```

(continues on next page)

(continued from previous page)

```
    [ -100.  -300.  -200.  -600.  -150.  -450.  -300.  -900.   100.   300.
    200.   600.   150.   450.   300.   900.]
    [ -100.  -200.  -300.  -600.  -150.  -300.  -450.  -900.   100.   200.
    300.   600.   150.   300.   450.   900.]
    [ -200.  -600.  -600. -1800.  -300.  -900.  -900. -2700.   200.   600.
    600.  1800.   300.   900.   900.  2700.]
    [  -75.  -150.  -150.  -300.  -175.  -350.  -350.  -700.    75.   150.
    150.   300.   175.   350.   350.   700.]
    [ -150.  -450.  -300.  -900.  -350. -1050.  -700. -2100.   150.   450.
    300.   900.   350.  1050.   700.  2100.]
    [ -150.  -300.  -450.  -900.  -350.  -700. -1050. -2100.   150.   300.
    450.   900.   350.   700.  1050.  2100.]
    [ -300.  -900.  -900. -2700.  -700. -2100. -2100. -6300.   300.   900.
    900.  2700.   700.  2100.  2100.  6300.]]
>>> print(b)
[ -198.5  -412.   -459.   -948.   -344.   -710.5  -781.  -1607.    198.5
    412.    459.    948.    344.    710.5   781.   1607. ]
```

jwst.wiimatch.lsq_optimizer.**pinv_solve**(*matrix*, *free_term*, *nimages*, *tol=None*)

Solves a system of linear equations

$$a \cdot c = b.$$

using Moore-Penrose pseudoinverse.

matrix [numpy.ndarray] A 2D array containing coefficients of the system.

free_term [numpy.ndarray] A 1D array containing free terms of the system of the equations.

nimages [int] Number of images for which the system is being solved.

tol [float, None, optional] Cutoff for small singular values for Moore-Penrose pseudoinverse. When provided, singular values smaller (in modulus) than `tol * |largest_singular_value|` are set to zero. When `tol` is `None` (https://docs.python.org/3/library/constants.html#None) (default), cutoff value is determined based on the type of the input `matrix` argument.

bkg_poly_coeff [numpy.ndarray] A 2D `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.n that holds the solution (polynomial coefficients) to the system. The solution is grouped by image.

```
>>> import wiimatch
>>> import numpy as np
>>> im1 = np.zeros((5, 5, 4), dtype=np.float)
>>> cbg = 1.32 * np.ones_like(im1)
>>> ind = np.indices(im1.shape, dtype=np.float)
>>> im3 = cbg + 0.15 * ind[0] + 0.62 * ind[1] + 0.74 * ind[2]
>>> mask = np.ones_like(im1, dtype=np.int8)
>>> sigma = np.ones_like(im1, dtype=np.float)
>>> a, b = wiimatch.lsq_optimizer.build_lsq_eqs([im1, im3], [mask, mask],
... [sigma, sigma], degree=(1,1,1), center=(0,0,0))
>>> wiimatch.lsq_optimizer.pinv_solve(a, b, 2)
array([[ -6.60000000e-01,  -7.50000000e-02,  -3.10000000e-01,
          3.33066907e-15,  -3.70000000e-01,   5.44009282e-15,
          7.88258347e-15,  -2.33146835e-15],
       [  6.60000000e-01,   7.50000000e-02,   3.10000000e-01,
```

(continues on next page)

(continued from previous page)

```
        -4.44089210e-15,   3.70000000e-01,  -4.21884749e-15,
        -7.43849426e-15,   1.77635684e-15]])
```

jwst.wiimatch.lsq_optimizer.**rlu_solve**(*matrix*, *free_term*, *nimages*)

Computes solution of a "reduced" system of linear equations

$$a' \cdot c' = b'.$$

using LU-decomposition. If the original system contained a set of linearly-dependent equations, then the "reduced" system is formed by dropping equations and unknowns related to the first image. The unknowns corresponding to the first image initially are assumed to be 0. Upon solving the reduced system, these unknowns are recomputed so that mean corection coefficients for all images are 0. This function uses `lu_solve` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_solve.html#scipy.linalg.lu_solve) and `lu_factor` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_factor.html#scipy.linalg.lu_factor) functions.

matrix [numpy.ndarray] A 2D array containing coefficients of the system.

free_term [numpy.ndarray] A 1D array containing free terms of the system of the equations.

nimages [int] Number of images for which the system is being solved.

bkg_poly_coeff [numpy.ndarray] A 2D `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.n
that holds the solution (polynomial coefficients) to the system. The solution is grouped by image.

```
>>> import wiimatch
>>> import numpy as np
>>> im1 = np.zeros((5, 5, 4), dtype=np.float)
>>> cbg = 1.32 * np.ones_like(im1)
>>> ind = np.indices(im1.shape, dtype=np.float)
>>> im3 = cbg + 0.15 * ind[0] + 0.62 * ind[1] + 0.74 * ind[2]
>>> mask = np.ones_like(im1, dtype=np.int8)
>>> sigma = np.ones_like(im1, dtype=np.float)
>>> a, b = wiimatch.lsq_optimizer.build_lsq_eqs([im1, im3], [mask, mask],
... [sigma, sigma], degree=(1, 1, 1), center=(0, 0, 0))
>>> wiimatch.lsq_optimizer.lu_solve(a, b, 2)
array([[ -6.60000000e-01,  -7.50000000e-02,  -3.10000000e-01,
         -1.19371180e-15,  -3.70000000e-01,  -1.62003744e-15,
         -1.10844667e-15,   5.11590770e-16],
       [  6.60000000e-01,   7.50000000e-02,   3.10000000e-01,
          1.19371180e-15,   3.70000000e-01,   1.62003744e-15,
          1.10844667e-15,  -5.11590770e-16]])
```

Utilities used by `wiimatch`

This module provides utility functions for use by `wiimatch` module.

Author Mihai Cara (contact: help@stsci.edu)

jwst.wiimatch.utils.**create_coordinate_arrays**(*image_shape*, *center=None*, *image2world=None*, *center_cs='image'*)

Create a list of coordinate arrays/grids for each dimension in the image shape. This function is similar to `numpy.indices`

(https://docs.scipy.org/doc/numpy/reference/generated/numpy.indices.html#numpy.indices) except it returns the list of arrays in reversed order. In addition, it can center image coordinates to a provided `center` and also convert image coordinates to world coordinates using provided `image2world` function.

image_shape [sequence of int] The shape of the image/grid.

center [iterable, None, optional] An iterable of length equal to the number of dimensions in `image_shape` that indicates the center of the coordinate system in **image** coordinates when `center_cs` is `'image'` otherwise center is assumed to be in **world** coordinates (when `center_cs` is `'world'`). When `center` is `None` (https://docs.python.org/3/library/constants.html#None) then `center` is set to the middle of the "image" as `center[i]=image_shape[i]//2`. If `image2world` is not `None` (https://docs.python.org/3/library/constants.html#None) and `center_cs` is `'image'`, then supplied center will be converted to world coordinates.

image2world [function, None, optional] Image-to-world coordinates transformation function. This function must be of the form `f(x, y,z,...)` and accept a number of arguments `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray) arguments equal to the dimensionality of images.

center_cs [{'image', 'world'}, optional] Indicates whether `center` is in image coordinates or in world coordinates. This parameter is ignored when `center` is set to `None` (https://docs.python.org/3/library/constants.html#None): it is assumed to be `False` (https://docs.python.org/3/library/constants.html#False). `center_cs` *cannot be* `'world'` when `image2world` is `None` (https://docs.python.org/3/library/constants.html#None) unless `center` is `None` (https://docs.python.org/3/library/constants.html#None).

coord_arrays [list] A list of `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html coordinate arrays each of `image_shape` shape.

eff_center [tuple] A tuple of coordinates of the effective center as used in generating coordinate arrays.

coord_system [{'image', 'world'}] Coordinate system of the coordinate arrays and returned `center` value.

```
>>> import wiimatch
>>> wiimatch.utils.create_coordinate_arrays((3,5,4))
((array([[[-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.]],
         [[-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.]],
         [[-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.],
          [-1.,  0.,  1.,  2.]]]),
    array([[[-2., -2., -2., -2.],
          [-1., -1., -1., -1.],
```

(continues on next page)

(continued from previous page)

```
          [ 0.,  0.,  0.,  0.],
          [ 1.,  1.,  1.,  1.],
          [ 2.,  2.,  2.,  2.]],
         [[-2., -2., -2., -2.],
          [-1., -1., -1., -1.],
          [ 0.,  0.,  0.,  0.],
          [ 1.,  1.,  1.,  1.],
          [ 2.,  2.,  2.,  2.]],
         [[-2., -2., -2., -2.],
          [-1., -1., -1., -1.],
          [ 0.,  0.,  0.,  0.],
          [ 1.,  1.,  1.,  1.],
          [ 2.,  2.,  2.,  2.]]]),
   array([[[-2., -2., -2., -2.],
          [-2., -2., -2., -2.],
          [-2., -2., -2., -2.],
          [-2., -2., -2., -2.],
          [-2., -2., -2., -2.]],
         [[-1., -1., -1., -1.],
          [-1., -1., -1., -1.],
          [-1., -1., -1., -1.],
          [-1., -1., -1., -1.],
          [-1., -1., -1., -1.]],
         [[ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.]]])), (1.0, 2.0, 2.0), u'image')
```

Python Module Index

j

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z