romancal

Release 0.1.dev108+g4897504.d20240426

Roman calibration pipeline developers <help@stsci.edu>

Apr 26, 2024

ROMANCAL PIPELINE

1 Introduction **3**

1.1 Installation . . . 3
1.2 Pipeline Levels . . . 5
1.3 Running the Pipeline . . . 8
1.4 Pipeline Steps . . . 9
1.5 Reference Files . . . 123
1.6 Parameters . . . 124
1.7 I/O File Conventions . . . 125
1.8 Error Propagation . . . 126
1.9 Data Products Information . . . 127
1.10 Associations . . . 132
1.11 Pipeline Modules . . . 136
1.12 Reference File Information . . . 139
1.13 STPIPE . . . 144
1.14 About datamodels . . . 168
1.15 Working with models . . . 168
1.16 Metadata . . . 170
1.17 Working with Roman datamodels and ASDF files . . . 172
1.18 ModelContainer . . . 177
1.19 Generating Static Previews . . . 180

Python Module Index **183**

Index **185**

genindex | modindex

Welcome to the documentation for the Roman calibration software, romancal. This package contains the Python software suite for the Roman Space Telescope (RST) calibration pipeline, which processes data from the Roman Wide-Field Instrument (WFI) by applying various corrections to produce science-ready, calibrated output products including fully calibrated individual exposures as well as high-level data products (mosaics, catalogs, etc.). The tools in this package allow users to run and configure the pipeline to custom process their Roman data. Additionally, the romancal package contains the interface to Roman datamodels, the recommended method of reading and writing Roman data files in Python.

If you have questions or concerns regarding the software, please contact the Roman Help desk at Roman Help Desk.

CHAPTER

ONE

INTRODUCTION

Welcome to the documentation for romancal. This package contains the calibration software for the WFI instrument on the Nancy Grace Roman Space Telescope.

The romancal package processes data by applying various corrections to produce science-ready output products, including fully calibrated individual exposures as well as high-level data products (e.g., mosaics and catalogs).

The tools in this package allow users to run and configure the pipeline steps to custom process the Roman data. Additionally, the romancal package contains the interface to the Roman datamodels, the recommended method of reading and writing Roman data files in Python.

1.1 Installation

Warning: Roman requires Python 3.10 or above and a C compiler for dependencies.

Warning: Linux and MacOS platforms are tested and supported. Windows is not currently supported.

Stable releases of the `romancal` package are registered at PyPI. The development version of `romancal` is installable from the Github repository.

The basic method of installing the roman calibration pipeline is to setup your environment and issue the command,

```
$ pip install romancal
```

1.1.1 Detailed Installation

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

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

- create a virtual environment;
- activate that environment;
- install the desired version of the `romancal` package into that environment.

Details are given below on how to do this for different types of installations, including tagged releases, DMS builds used in operations, and development versions. Note that, although you can use any python environment management system that you are familiar with, below we will be using `conda` (remember that all conda operations must be done from within a bash shell).

Installing latest releases

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

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

You can also install a specific version (from `romancal 0.1.0` onward):

```
$ conda create -n <env_name> python
$ conda activate <env_name>
$ pip install romancal==0.5.0
```

Installing the development version from Github

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

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

Installing for Developers

If you want to be able to work on and test the source code with the `romancal` package, the high-level procedure to do this is to first create a conda environment using the same procedures outlined above, but then install your personal copy of the code overtop of the original code in that environment. Again, this should be done in a separate conda environment from any existing environments that you may have already installed with released versions of the `romancal` package.

As usual, the first two steps are to create and activate an environment:

```
$ conda create -n <env_name> python
$ conda activate <env_name>
```

To install your own copy of the code into that environment, you first need to fork and clone the `romancal` repo:

```
$ cd <where you want to put the repo>
$ git clone https://github.com/spacetelescope/romancal
$ cd romancal
```

Note: `python setup.py install` and `python setup.py develop` commands do not work.

Install from your local checked-out copy as an "editable" install:

```
$ pip install -e .
```

If you want to run the unit or regression tests and/or build the docs, you can make sure those dependencies are installed as well:

```
$ pip install -e '.[test]'
$ pip install -e '.[docs]'
$ pip install -e '.[test,docs]'
```

Need other useful packages in your development environment?

```
$ pip install ipython pytest-xdist
```

1.1.2 Calibration References Data System (CRDS) Setup

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

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

1.2 Pipeline Levels

1.2.1 Exposure Level Processing

Class
romancal.pipeline.ExposurePipeline

Alias
exposure_pipeline

The `ExposurePipeline` applies detector-level corrections to given exposure types (imaging, prism, and grism.). It is applied to one exposure at a time. It is sometimes referred to as "ramps-to-slopes" processing, because the input raw data are in the form of ramps 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 `ExposurePipeline` pipeline is shown in the table below.

Step	WFI-Image	WFI-Prism	WFI-Grism
dq_init	✓	✓	✓
saturation	✓	✓	✓
refpix	✓	✓	✓
linearity	✓	✓	✓
dark_current	✓	✓	✓
ramp_fitting	✓	✓	✓
assign_wcs	✓	✓	✓
flatfield	✓		
photom	✓		
source_detection	✓		
tweakreg	✓		

Arguments

The `exposure` pipeline has an optional argument:

```
--use_ramp_jump_detection  boolean  default=True
```

When set to `True`, the pipeline will perform *jump* detection as a part of the ramp fitting step. The data at this stage of the pipeline are still in the form of the original 3D ramps (ngroups x ncols x nrows) and have had all of the detector-level correction steps applied to it, up to but not including the detection and flagging of Cosmic-Ray (CR) hits within each ramp (integration). For this case the *jump* module in *ramp_fitting* will update the dq array with the CR hits (jumps) that are identified in the step.

Inputs

3D raw data

Data model
`RampModel`

File suffix
_uncal

The input to the `ExposurePipeline` is a single raw exposure, e.g. "r0008308002010007027_06311_0019_WFI01_uncal.asdf", which contains the original raw data from all of the detector readouts in the exposure (ngroups x ncols x nrows).

Note that in the operational environment, the input will be in the form of a `RawScienceModel`, which only contains the 3D array of detector pixel values, along with some optional extensions. When such a file is loaded into the pipeline, it is immediately converted into a `RampModel`, and has all additional data arrays for errors and Data Quality flags created and initialized.

Outputs

2D Image model

Data model
`ImageModel`

File suffix
_cal

Result of applying all pipeline steps up through the *tweakreg* step is to produce calibrated data with the image WCS aligned to Gaia, and is 2D image data, which will have one less data dimensions as the input raw 3D data. In addition to being a 2-dimensional image the output from the pipeline has the *reference pixels* removed from the edges of the science array and saved as additional 3D arrays.

1.2.2 High Level Image Processing

Class
romancal.pipeline.HighLevelPipeline

Alias
highlevel_pipeline

The `HighLevelPipeline` applies corrections to an overlapping group of images and is setup to process only imaging observations. This pipeline is used to determine a common background, *skymatch*, detect pixels the are not consistent with the other datasets, *outlier_detection*, and resample the image to a single undistorted image, *resample*.

The list of steps applied by the `HighLevelPipeline` pipeline is shown in the table below.

Step	WFI-Image	WFI-Prism	WFI-Grism
skymatch	✓		
outlier_detection	✓		
resample	✓		

Arguments

The `highlevel` pipeline has no optional arguments:

You can see the options for strun using:

strun –help roman_hlp

and this will list all the strun options all well as the step options for the roman_hlp.

Inputs

An association of 2D calibrated image data

Data model
`WfiImage`

File suffix
_cal

The input to the `HighLevelPipeline` is a group of calibrated exposures, e.g. "r0008308002010007027_06311_0019_WFI01_cal.asdf", which contains the calibrated data for the the exposures. The most convenient way to pass the list of exposures to be processed with the high level pipeline is to use an association. Instructions on how to create an input association an be found at *asn_from_list*.

Outputs

2D Image (MosaicModel)

Data model
`WfiMosaic`

File suffix
_i2d

Result of applying all the high level pipeline steps up through the *resample* step is to produce data background corrected and cleaned of outliers and resampled to a distortion free grid. This is 2D image data, with additional attributes for the mosaicing information.

1.3 Running the Pipeline

Note: The Roman calibration code does not yet use CRDS parameters and it is best to disable the search for these parameters to avoid unexpected issues. You can globally disable this with the environment variable STPIPE_DISABLE_CRDS_STEPPARS In the Bash shell you can issue the command,

export STPIPE_DISABLE_CRDS_STEPPARS=True

You can also add this parameter the the strun command,

strun –disable-crds-steppar roman_elp <file_name>

1.3.1 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 <pipeline_name or class_name> input_file>
```

The first argument to `strun` must be one of either a pipeline name, python class of the step or pipeline to be run. The second argument to `strun` is the name of the input data file to be processed. For a list of all the options available for `strun`, please read the STPIPE Documentation.

For example, the exposure level pipeline is implemented by the class *romancal.pipeline.ExposurePipeline*. The command to run this pipeline is:

```
$ strun romancal.pipeline.ExposurePipeline r0008308002010007027_06311_0019_WFI01_uncal.
↪asdf
```

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

```
$ strun roman_elp r0008308002010007027_06311_0019_WFI01_uncal.asdf
```

The high level pipeline can be run in a similar manner and is implemented using the class *romancal.pipeline.HighLevelPipeline*. The command to run this pipeline is:

```
$ strun romancal.pipeline.HighLevelPipeline r0008308002010007027_asn.json
```

An important point is that the high level pipeline needs multiple exposures to run correctly. The most convenient method to supply the input is to use an association. Instructions on how to create an input association an be found at *asn_from_list*.

The high level pipeline also has an alias, `roman_hlp`, and can be run as

```
$ strun roman_hlp r0008308002010007027_asn.json
```

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

1.3.2 From the Python Prompt

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

The `call` method creates a new instance of the class and runs the pipeline or step. Optional parameter settings can be specified by via keyword arguments or supplying a parameter file. Some examples are shown below.

For the exposure pipeline and steps,

```
from romancal.pipeline import ExposurePipeline
result = ExposurePipeline.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf')

from romancal.linearity import LinearityStep
result = LinearityStep.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf')
```

One difference between the high level pipeline and the exposure level pipeline is that the high level pipeline is generally designed to run on multiple overlapping exposures. To achieve that the input to the pipeline is a list of images, usually an association. For the high level pipeline and steps,

```
from romancal.pipeline import HighLevelPipeline
result = ExposurePipeline.call('r0000101001001001001_asn.json')

from romancal.skymatch import SkyMatchStep
result = SkyMatchStep.call('r0000101001001001001_asn.json')
```

For more information, see *Execute via call()*

For details on the different ways to run a pipeline step, see the *Configuring a Step* page.

1.4 Pipeline Steps

1.4.1 Data Quality (DQ) Initialization

Description

The Data Quality (DQ) initialization step in the calibration pipeline populates the DQ mask for the input dataset. Flag values from the appropriate static mask ("MASK") reference file in CRDS are copied into the "PIXELDQ" array of the input dataset, because it is assumed that flags in the mask reference file pertain to problem conditions that affect all groups for a given pixel.

The actual process consists of the following steps:

- Determine what MASK reference file to use via the interface to the bestref utility in CRDS.
- Copy the input product into a RampModel (if it isn't already) for processing through pipeline. This will create "pixeldq" and "groupdq" arrays (if they don't already exist).

- Propagate the DQ flags from the reference file DQ array to the science data "PIXELDQ" array using numpy's `bitwise_or` function.

Note that when applying the `dq_init` step to guide star data, the flags from the MASK reference file are propagated into the guide star dataset "dq" array, instead of the "pixeldq" array. The step identifies guide star data based on the following exposure type (exposure.type keyword attribute) values: WFI_WIM_ACQ, WFI_WIM_TRACK, WFI_WSM_ACQ1, WFI_WSM_ACQ2, WFI_WSM_TRACK.

Step Arguments

The Data Quality Initialization step has no step-specific arguments.

Reference Files

The `dq_init` step uses a MASK reference file.

A list of the allowed DQ values are:

Flags for the DQ, PIXELDQ, and GROUPDQ Arrays.

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	GW_AFFECTED_DATA	Data affected by the GW read window
5	32	PERSISTENCE	High persistence (was RESERVED_2)
6	64	AD_FLOOR	Below A/D floor (0 DN, was RESERVED_3)
7	128	OUTLIER	Detected as outlier in coadded image
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
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	RESERVED_5	
27	134217728	RESERVED_6	
28	268435456	UNRELIABLE_RESET	Sensitive to reset anomaly
29	536870912	RESERVED_7	
30	1073741824	OTHER_BAD_PIXEL	A catch-all flag
31	2147483648	REFERENCE_PIXEL	Pixel is a reference pixel

MASK Reference File

reftype
MASK

Data model
`MaskRefModel`

The MASK reference file contains pixel-by-pixel DQ flag values that indicate problem conditions.

Reference Selection Keywords for MASK

CRDS selects appropriate MASK references based on the following keywords. MASK is not applicable for instruments not in the table.

Instrument	Metadata
WFI	instrument, detector, date, time

Standard ASDF metadata

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

Attribute	Fully Qualified Path
author	model.meta.author
model_type	model.meta.model_type
date	model.meta.date
description	model.meta.description
instrument	model.meta.instrument.name
reftype	model.meta.reftype
telescope	model.meta.telescope
useafter	model.meta.useafter

NOTE: More information on standard required attributes can be found here: *Standard ASDF metadata*

Type Specific Keywords for MASK

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

Attribute	Fully qualified path	Instruments
detector	model.meta.instrument.detector	WFI

Reference File Format

MASK reference files are ASDF format, with one data object. The format and content of the file is as follows:

Data	Object Type	Dimensions	Data type
dq	NDArray	4096 x 4096	uint32

The values in the `dq` array give the per-pixel flag conditions that are to be propagated into the science exposure's `pixeldq` array. The dimensions of the `dq` array should be equal to the number of columns and rows in a full-frame readout of a given detector, including reference pixels.

The ASDF file contains a single dq array.

romancal.dq_init Package

Classes

DQInitStep([name, parent, config_file, ...])	Initialize the Data Quality extension from the mask reference file.

DQInitStep

class romancal.dq_init.**DQInitStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Initialize the Data Quality extension from the mask reference file.

The dq_init step initializes the pixeldq attribute of the input datamodel using the MASK reference file. For some Guiding and Image model types, initalize the dq attribute of the input model instead. The dq attribute of the MASK model is bitwise OR'd with the pixeldq (or dq) attribute of the input model.

Create a `Step` instance.

Parameters

- **name** (*str, 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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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)	Perform the dq_init calibration step

Attributes Documentation

reference_file_types: ClassVar = ['mask']

Methods Documentation

process(*input*)

Perform the dq_init calibration step

Parameters
input (*Roman datamodel*) – input roman datamodel

Returns
output_model – result roman datamodel

Return type
Roman datamodel

Class Inheritance Diagram

1.4.2 Saturation Detection

Description

The `saturation` step flags pixels at or below the A/D floor or above the saturation threshold. Pixels values are flagged as saturated if the pixel value is larger than the defined saturation threshold. Pixel values are flagged as below the A/D floor if they have a value of zero DN.

This step examines the data group-by-group, comparing the pixel values in the data array with defined saturation thresholds for each pixel. When it finds a pixel value in a given group that is above the saturation threshold (high saturation) times a dilution factor, it sets the "SATURATED" flag in the corresponding location of the "groupdq" array in the science exposure. When it finds a pixel in a given group that has a zero or negative value (below the A/D floor), it sets the "AD_FLOOR" and "DO_NOT_USE" flags in the corresponding location of the "groupdq" array in the science exposure For the saturation case, it also flags all subsequent groups for that pixel as saturated. For example, if there are 10 groups and group 7 is the first one to cross the saturation threshold for a given pixel, then groups 7 through 10 will all be flagged for that pixel.

Pixels with thresholds set to NaN or flagged as "NO_SAT_CHECK" in the saturation reference file have their thresholds set to the 16-bit A-to-D converter limit of 65535 and hence will only be flagged as saturated if the pixel reaches that hard limit in at least one group. The "NO_SAT_CHECK" flag is propagated to the PIXELDQ array in the output science data to indicate which pixels fall into this category.

The "dilution factor" is intended to account for the fact that Roman downlinks resultants to the ground, which are usually averages over several reads. The saturation threshold corresponds to the number of counts at which a detector pixel saturates. Because a resultant is the average of a number of reads, later reads in a resultant can saturate, but if earlier reads are unsaturated, the value of the resultant can fall under the saturation threshold. The dilution factor varies resultant-by-resultant and is given by $< t > /max(t)$ for all times t entering a resultant.

Step Arguments

The `saturation` step has no step-specific arguments.

Reference Files

The `saturation` step uses a SATURATION reference file.

SATURATION Reference File

REFTYPE
SATURATION

Data model
`SaturationRefModel`

The SATURATION reference file contains pixel-by-pixel saturation threshold values.

Reference Selection Keywords for SATURATION

CRDS selects appropriate SATURATION references based on the following metadata attributes. All attributes used for file selection are *required*.

Instrument	Keywords
WFI	instrument, detector, date, time

Standard ASDF metadata

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

Attribute	Fully Qualified Path
author	model.meta.author
model_type	model.meta.model_type
date	model.meta.date
description	model.meta.description
instrument	model.meta.instrument.name
reftype	model.meta.reftype
telescope	model.meta.telescope
useafter	model.meta.useafter

NOTE: More information on standard required attributes can be found here: *Standard ASDF metadata*

Type Specific Keywords for SATURATION

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

Keyword	Data Model Name	Instrument
detector	model.meta.instrument.detector	WFI

Reference File Format

SATURATION reference files are ASDF format, with two data objects. The format and content of the file is as follows:

Data	Object Type	Dimensions	Data type
data	NDArray	4096 x 4096	float32
dq	NDArray	4096 x 4096	uint32

The values in the `data` array give the saturation threshold in units of DN for each pixel.

The ASDF file contains two data arrays.

romancal.saturation Package

Classes

SaturationStep([name, parent, config_file, ...])	This Step sets saturation flags.

SaturationStep

class romancal.saturation.**SaturationStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

This Step sets saturation flags.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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: ClassVar = ['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

1.4.3 Reference Pixel Correction

Description

Overview

The `refpix` step corrects for additional signal from the electronics using the reference pixels.

Reference Pixels in Data

The WFI has two sets of reference pixels: a 4-pixel border of reference pixels around the science pixels, and the Amp33 reference pixels which are a 4096 x 128 section of the detector adjacent to the border pixels.

In the data files, the storage location of the reference pixels depends on level of processing.

A Level 1, uncalibrated image has one array that contains both the science pixels and the border reference pixel, and a separate array for the Amp33 pixels.

A RampModel, which is created during the `dq_init` step, represents a dataset at any intermediate step between Level 1 and the final Level 2 image. Like the Level 1 file, RampModels also contain an array with both the science and border reference pixels, and another with the Amp 33 reference pixels. In addition to these arrays, there are four more arrays that contain the original border reference pixels (top, bottom, left, and right), and an additional four for their DQ arrays. The border pixels are copied during the `dq_init`, so they reflect the original state of the border pixels before any calibration. The border pixels that are still attached to the science data in the RampModel will later be discarded when the Level 2 image is created. Note that the border reference pixel arrays each include the overlap regions in the corners, so that each slice contains the full span of border pixels at the top, bottom, left, or right.

In the Level 2, calibrated image, the data array only contains the science pixels. The border reference pixels are trimmed from this image duing ramp_fit. The additional arrays for the original border reference pixels (which are 3D) and their DQ arrays, and the Amp 33 reference pixels, remain in the Level 2 file.

Discretization bias & reference pixel correction

The analog-to-digital conversion in the Roman electronics performs an integer floor operation that biases the downlinked signal low relative to the actual number of photons observed by the instrument. The equation for this "discretization bias" is given by:

$$\text{bias} = -0.5 - 0.5\frac{N-1}{N},$$

in units of counts, where N is the number of reads entering into a particular resultant. This is a small effect. The constant -0.5 term is degenerate with the pedestal and has no effect on ramp slopes and therefore on the primary astronomical quantity of interest. The second term, however, depends on the number of reads in a resultant and may vary from resultant to resultant in Roman. This, if uncorrected, can lead to a bias in the fluxes we derive from Roman data for sources.

However, we need take no special action to correct for this effect. The reference pixels are affected by the discretization bias in the same way as the science pixels, and so when the reference pixels are subtracted (roughly speaking!) from the science pixels, this bias cancels. Exactly when this cancellation occurs depends on the details of the reference pixel correction step. Presently the reference pixel correct includes a component that removes trends across each amplifier and frame using the reference pixels at the top and bottom of the amplifier. This removes the discretization bias.

We note that even if the discretization bias were not removed at the reference pixel correction stage, it could be corrected at the dark subtraction step. Provided that dark reference images are processed through the usual reference pixel correction step, they will have the same biases present in the reference-pixel-corrected images. We have decided to perform the dark subtraction of Roman images via subtracting precomputed images for each MA table rather than scaling a fixed dark rate image by the mean time of each resultant. These precomputed dark images will contain not only the dark current but also electronic effects like the discretization bias. However, it is better to correct this effect during the reference pixel correction so that the dark reference images better represent the dark current and can be more easily used to compute Poisson uncertainties stemming from dark current.

1.4.4 Linearity Correction

Description

Assumptions

It is assumed that the saturation step has already been applied to the input data, so that saturation flags are set in the GROUPDQ array of the input science data.

Algorithm

The linearity step applies the "classic" linearity correction adapted from the HST WFC3/IR linearity correction routine, correcting science data values for detector non-linearity. The correction is applied pixel-by-pixel, group-by-group, integration-by-integration within a science exposure.

The correction is represented by an nth-order polynomial for each pixel in the detector, with n+1 arrays of coefficients read from the linearity reference file.

The algorithm for correcting the observed pixel value in each group of an integration is currently of the form:

$$F_{\rm c} = c_0 + c_1F + c_2F^2 + c_3F^3 + ... + c_nF^n$$

where F is the observed counts (in DN), c_n are the polynomial coefficients, and $F_{\rm c}$ is the corrected counts. There is no limit to the order of the polynomial correction; all coefficients contained in the reference file will be applied.

Upon successful completion of the linearity correction, "cal_step" in the metadata is set to "COMPLETE".

Special Handling

- Pixels having at least one correction coefficient equal to NaN will not have the linearity correction applied and the DQ flag "NO_LIN_CORR" is added to the science exposure PIXELDQ array.
- Pixels that have the "NO_LIN_CORR" flag set in the DQ array of the linearity reference file will not have the correction applied and the "NO_LIN_CORR" flag is added to the science exposure PIXELDQ array.
- Pixel values that have the "SATURATED" flag set in a particular group of the science exposure GROUPDQ array will not have the linearity correction applied to that group. Any groups for that pixel that are not flagged as saturated will be corrected.

The ERR array of the input science exposure is not modified.

The flags from the linearity reference file DQ array are propagated into the PIXELDQ array of the science exposure using a bitwise OR operation.

Arguments

The linearity correction has no step-specific arguments.

Reference File Types

The `linearity` step uses a LINEARITY reference file.

LINEARITY Reference File

REFTYPE
LINEARITY

Data model
`LinearityModel`

The LINEARITY reference file contains pixel-by-pixel polynomial correction coefficients.

Reference Selection Keywords for LINEARITY

CRDS selects appropriate LINEARITY references based on the following keyword attributes. All keyword attributes used for file selection are *required*.

Instrument	Keyword Attributes
WFI	instrument, detector, date, time

Standard ASDF metadata

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

Attribute	Fully Qualified Path
author	model.meta.author
model_type	model.meta.model_type
date	model.meta.date
description	model.meta.description
instrument	model.meta.instrument.name
reftype	model.meta.reftype
telescope	model.meta.telescope
useafter	model.meta.useafter

NOTE: More information on standard required attributes can be found here: *Standard ASDF metadata*

Type Specific Attributes for LINEARITY

In addition to the standard reference file attributes listed above, the following attributes are *required* in LINEARITY reference files, because they are used as CRDS selectors (see linearity_selectors):

Attribute	Fully qualified path	Instruments
detector	model.meta.intstrument.detector	WFI

Reference File Format

LINEARITY reference files are ASDF format, with 2 data arrays. The format and content of the file is as follows:

Data	Array Type	Dimensions	Data type
coeffs	NDArray	ncols x nrows x ncoeffs	float32
dq	NDArray	ncols x nrows	uint32

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.

romancal.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 romancal.linearity.**LinearityStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

LinearityStep: This step performs a correction for non-linear detector response, using the "classic" polynomial method.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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: ClassVar = ['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

1.4.5 Dark Current Subtraction

Description

Assumptions

It is assumed that the input science data have had the zero group (or bias) subtracted. Accordingly, the dark reference data should have their own group zero subtracted from all groups.

Algorithm

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

The current implementation uses dark reference files that are matched to the MA table entry in the exposure metadata. Note that the data reference file for a science group (SCI) is named `data`. The dark data are then subtracted, group-by-group, from the science exposure groups, in which each SCI group of the dark data is subtracted from the corresponding SCI group of the science data.

The ERR arrays of the science data are not modified.

The DQ flags from the dark reference file are propagated into science exposure PIXELDQ array using a bitwise OR operation.

Upon successful completion of the dark subtraction the cal_step attribute is set to "COMPLETE".

Special Handling

Any pixel values in the dark reference data that are set to NaN will have their values reset to zero before being subtracted from the science data, which will effectively skip the dark subtraction operation for those pixels.

Step Arguments

The dark current step has one step-specific argument:

- `--dark_output`

If the `dark_output` argument is given with a filename for its value, the frame-averaged dark data that are created within the step will be saved to that file.

Reference File

The `dark` step uses a DARK reference file.

DARK Reference File

REFTYPE
DARK

Data models
`DarkRefModel`

The DARK reference file contains pixel-by-pixel and frame-by-frame dark current values for a given detector readout mode.

Reference Selection Keyword Attributes for DARK

CRDS selects appropriate DARK references based on the following keyword attributes. DARK is not applicable for instruments not in the table.

Instrument	Keyword Attributes
WFI	instrument, detector, date, time

Standard ASDF metadata

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

Attribute	Fully Qualified Path
author	model.meta.author
model_type	model.meta.model_type
date	model.meta.date
description	model.meta.description
instrument	model.meta.instrument.name
reftype	model.meta.reftype
telescope	model.meta.telescope
useafter	model.meta.useafter

NOTE: More information on standard required attributes can be found here: *Standard ASDF metadata*

Type Specific Keyword Attributes for DARK

In addition to the standard reference file keyword attributes listed above, the following keyword attributes are *required* in DARK reference files, because they are used as CRDS selectors (see *Reference Selection Keyword Attributes for DARK*):

Attribute	Fully qualified path	Instruments
detector	model.meta.instrument.detector	WFI

Reference File Format

DARK reference files are ASDF format, with 3 data arrays. The format and content of the file is as follows (see `DarkRefModel`):

Data	Array Type	Dimensions	Data type
data	NDArray	4096 x 4096 x ngroups	float32
err	NDArray	4096 x 4096 x ngroups	float32
dq	NDArray	4096 x 4096	uint32

The ASDF file contains a single set of data, err, and dq arrays.

romancal.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 romancal.dark_current.**DarkCurrentStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

DarkCurrentStep: Performs dark current correction by subtracting dark current reference data from the input science data model.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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: ClassVar = ['dark']

spec

```
dark_output = output_file(default = None) # Dark corrected model
```

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

1.4.6 Jump Detection

Jump Detection for Uneven Ramps

Note: This version of jump detection will work on both unevenly-spaced and evenly-spaced ramps. So there is no need to worry about which type of of ramp you have.

Description

This "step" is actually integrated into the *unevenly-spaced ramp fitting algorithm*.

Jump Detection for Even Ramps

Warning: This step can only be run on evenly-spaced ramps. Using turning this step will turn off the jump detection algorithm integrated into ramp fitting.

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 (`refpix`) and non-linearity correction (`linearity`) have been applied, so that the input data ramps do not have any non-linearities or noise above the modeled Poission and read noise due to instrumental effects. The absence of any of these preceding corrections or residual non-linearities or noise can lead to the false detection of jumps in the ramps, due to departure from linearity.

The `jump` step will automatically skip execution if the input data contain fewer than 3 groups in the integration, because the baseline algorthim requires two first differences to work.

Algorithm

This routine detects jumps in an exposure by looking for outliers in the up-the-ramp signal for each pixel in the integration within an input exposure. On output, the GROUPDQ array is updated with the DQ flag "JUMP_DET" to indicate the location of each jump that was found. In addition, any pixels that have non-positive or NaN values in the gain reference file will have DQ flags "NO_GAIN_VALUE" and "DO_NOT_USE" set in the output PIXELDQ array. The SCI and ERR arrays of the input data are not modified.

The current implementation uses the two-point difference method described in Anderson&Gordon2011_.

Two-Point Difference Method

The two-point difference method is applied to the integration as follows:

- Compute the first differences for each pixel (the difference between adjacent groups)
- Compute the clipped (dropping the largest difference) 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 difference.
- Compute the "difference ratio" as the difference between the first differences of each group and the median, divided by the expected noise.
- If the largest "difference ratio" is greater than the rejection threshold, flag the group corresponding to that ratio as having a jump.
- If a jump is found in a given pixel, iterate the above steps with the jump-impacted group excluded, looking for additional lower-level jumps that still exceed the rejection threshold.
- Stop iterating on a given pixel when no new jumps are found or only one difference remains.
- If the there are only three differences (four groups), the standard median is used rather than the clipped median.
- If there are only two differences (three groups), the smallest one is compared to the larger one and if the larger one is above a threshold, it is flagged as a jump.

Note that any ramp values flagged as SATURATED in the input GROUPDQ array are not used in any of the above calculations and hence will never be marked as containing a jump.

Multiprocessing

This step has the option of running in multiprocessing mode. In that mode it will split the input data cube into a number of row slices based on the number of available cores on the host computer and the value of the max_cores input parameter. By default the step runs on a single processor. At the other extreme if max_cores is set to 'all', it will use all available cores (real and virtual). Testing has shown a reduction in the elapsed time for the step proportional to the number of real cores used. Using the virtual cores also reduces the elasped time but at a slightly lower rate than the real cores.

If multiprocessing is requested the input cube will be divided into a number of slices in the row dimension (with the last slice being slightly larger, if needed). The slices are then sent to twopoint_difference.py by detect_jumps. After all the slices have finished processing, detect_jumps assembles the output group_dq cube from the slices.

Arguments

The `jump` step has five optional arguments that can be set by the user:

- `--rejection_threshold`: A floating-point value that sets the sigma threshold for jump detection for ramps having 5 or more groups.. In the code sigma is determined using the read noise from the read noise reference file and the Poisson noise (based on the median difference between samples, and the gain reference file). Note that any noise source beyond these two that may be present in the data will lead to an increase in the false positive rate and thus may require an increase in the value of this parameter. The default value of 4.0 for the rejection threshold will yield 6200 false positives for every million pixels, if the noise model is correct.
- `--three_group_rejection_threshold`: A floating-point value that sets the sigma threshold for jump detection for ramps having exactly 3 groups. The default value is 6.0
- `--four_group_rejection_threshold`: A floating-point value that sets the sigma threshold for jump detection for ramps having exactly 4 groups. The default value is 5.0
- `--maximum_cores`: The fraction of available cores that will be used for multi-processing in this step. The default value is 'none' which does not use multi-processing. The other options are 'quarter', 'half', and 'all'. Note that these fractions refer to the total available cores and on most CPUs these include physical and virtual cores. The clock time for the step is reduced almost linearly by the number of physical cores used on all machines. For example, on an Intel CPU with six real cores and 6 virtual cores setting maximum_cores to 'half' results in a decrease of a factor of six in the clock time for the step to run. Depending on the system the clock time can also decrease even more with maximum_cores is set to 'all'.
- `--flag_4_neighbors`: If set to True (default is True) it will cause the four perpendicular neighbors of all detected jumps to be flagged as a jump. This is needed because of the inter-pixel capacitance (IPC) causing a small jump in the neighbors. The small jump might be below the rejection threshold but will affect the slope determination of the pixel. The step will take about 40% longer to run when this is set to True.
- `--max_jump_to_flag_neighbors`: A floating point value in units of sigma that limits the flagging of neighbors. Any jump above this cutoff will not have its neighbors flagged. The concept is that the jumps in neighbors will be above the rejection-threshold and thus be flagged as primary jumps. The default value is 1000.
- `--min_jump_to_flag_neighbors`: A floating point value in units of sigma that limits the flagging of neighbors of marginal detections. Any primary jump below this value will not have its neighbors flagged. The goal is to prevent flagging jumps that would be too small to significantly affect the slope determination. The default value is 10.
- `--use_ramp_jump_detection`: See the description in *ramp fitting*.

Reference File Types

The `jump` step uses two reference files: GAIN and READNOISE. The GAIN reference file is used to temporarily convert pixel values in the `jump` step from units of DN to electrons. The READNOISE reference file is used in estimating the expected noise in each pixel. Both are necessary for proper computation of noise estimates within the `jump` step.

GAIN

READNOISE

romancal.jump Package

Classes

JumpStep([name, parent, config_file, ...])	JumpStep: Performs CR/jump detection.

JumpStep

class romancal.jump.**JumpStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

JumpStep: Performs CR/jump detection. The 2-point difference method is applied.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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: ClassVar = ['gain', 'readnoise']

spec

```
rejection_threshold = float(default=180.0,min=0) # CR sigma rej thresh
three_group_rejection_threshold = float(default=185.0,min=0) # CR sigma rej␣
↪thresh
four_group_rejection_threshold = float(default=190.0,min=0) # CR sigma rej␣
↪thresh
maximum_cores = option('none', 'quarter', 'half', 'all', default='none') # max␣
↪number of processes to create
flag_4_neighbors = boolean(default=True) # flag the four perpendicular␣
↪neighbors of each CR
max_jump_to_flag_neighbors = float(default=1000) # maximum jump sigma that will␣
↪trigger neighbor flagging
min_jump_to_flag_neighbors = float(default=10) # minimum jump sigma that will␣
↪trigger neighbor flagging
min_sat_area = float(default=1.0) # minimum required area for the central␣
↪saturation of snowballs
min_jump_area = float(default=5.0) # minimum area to trigger large events␣
↪processing
expand_factor = float(default=2.0) # The expansion factor for the enclosing␣
↪circles or ellipses
use_ellipses = boolean(default=False) # Use an enclosing ellipse rather than a␣
↪circle for MIRI showers
sat_required_snowball = boolean(default=True) # Require the center of snowballs␣
↪to be saturated
expand_large_events = boolean(default=False) # must be True to trigger snowball␣
↪and shower flagging
use_ramp_jump_detection = boolean(default=True) # Use jump detection during␣
↪ramp fitting
```

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

1.4.7 Ramp Fitting

Description: Optimized Least-squares with Uneven Ramps

This step determines the mean count rate, in units of counts per second, for each pixel by performing a linear fit to the data in the input file. The fit is done using the "ordinary least squares" method. The fit is performed independently for each pixel. There can be up to two output files created by the step. The primary output file ("rate") contains the slope at each pixel. A second, optional output product is also available, containing detailed fit information for each pixel. The two types of output files are described in more detail below.

The count rate for each pixel is determined by a linear fit to the cosmic-ray-free and saturation-free ramp intervals for each pixel; hereafter this interval will be referred to as a "segment." There are two algorithms used: Optimal Least-Square ('ols') and Optimal Least-Square for Uneven ramps ('ols_cas22'). The 'ols' algorithm is the one used by JWST and is further *described here*.

The 'ols_22' algorithm is based on Casertano et al, STScI Technical Document, 2022. The implementation is what is described in this document.

Segments

Segments are determined using the 3-D GROUPDQ array of the input data set, under the assumption that the jump step will have already flagged CR's. Segments are terminated where saturation flags are found.

The ramp fitting step is also where the *reference pixels* are trimmed, resulting in a smaller array being passed to the subsequent steps.

Special Cases

If the input dataset has only a single resultant, no fit is determined, giving that resultant a weight of zero.

All Cases

For all input datasets, including the special cases described above, arrays for the primary output (rate) product are computed as follows.

After computing the slopes for all segments for a given pixel, the final slope is determined as a weighted average from all segments, and is written as the primary output product. In this output product, the 3-D GROUPDQ is collapsed into 2-D, merged (using a bitwise OR) with the input 2-D PIXELDQ, and stored as a 2-D DQ array.

Slope and Variance Calculations

Slopes and their variances are calculated for each segment, and for the entire exposure. As defined above, a segment is a set of contiguous resultants where none of the resultants are saturated or cosmic ray-affected. The appropriate slopes and variances are output to the primary 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', 'o' for segment, or overall (for the entire dataset), respectively.

Dark current

Ramp fitting receives dark-subtracted ramps as input, but the Poisson noise in the dark current contributes to the noise in the ramps. So we need to add the dark current back into the ramps before ramp fitting, and then subtract it off again from the ramp fits.

Dark current in Roman is low and this makes very little difference for most pixels.

Optimal Weighting Algorithm

The slope of each segment is calculated using the least-squares method with optimal weighting, as described by Casertano et al, STScI Technical Document, 2022. Optimal weighting determines the relative weighting of each sample when calculating the least-squares fit to the ramp. When the data have low signal-to-noise ratio S, the data are read noise dominated and equal weighting of samples is the best approach. In the high signal-to-noise regime, data are Poisson-noise dominated and the least-squares fit is calculated with the first and last samples. In most practical cases, the data will fall somewhere in between, where the weighting is scaled between the two extremes.

The signal-to-noise ratio S used for weighting selection is calculated from the last sample as:

$$S_{max} = S_{last} - S_{first}$$
$$S = \frac{S_{max}}{\sqrt{(read\_noise)^2 + S_{max}}},$$

where S_{max} is the maximum signal in electrons with the pedestal removed.

The weighting for a sample i is given as:

$$w_i = \frac{(1+P) \times N_i}{1 + P \times N_i} |\bar{t}_i - \bar{t}_{mid}|^P ,$$

where t_{mid} is the time midpoint of the sequence, N_i is the number of contributing reads, and P is the exponent applied to weights, determined by the value of S. Fixsen et al. 2000 found that defining a small number of P values to apply to values of S was sufficient; they are given as:

Minimum S	Maximum S	P
0	5	0
5	10	0.4
10	20	1
20	50	3
50	100	6
100		10

Segment-Specific Computations

The segment fitting implementation is based on Section 5 of Casertano et. al. 2022. Segments which have only a single resultant, no fitting is performed.

A set of auxiliary quantities are computed as follows:

$$F0 = \sum_{i=0}^{N-1} W_i$$
$$F1 = \sum_{i=0}^{N-1} W_i \bar{t}_i$$
$$F2 = \sum_{i=0}^{N-1} W_i \bar{t}_i^2$$

The denominator, D, is calculated as a single two-dimensional array:

$$D = F2 \cdot F0 - F1^2$$

The resultant coefficients, K_i, are computed as N two dimensional arrays:

$$K_i = (F0 \cdot \bar{t}_i - F1) \cdot W_i / D$$

The estimated slope, $\hat{F}$, is computed as a sum over the resultants R_i and the coefficients K_i:

$$\hat{F} = \sum_i K_i R_i$$

The read-noise component V_R of the slope variance is computed as:

$$V_R = \sum_{i=0}^{N-1} K_i^2 \cdot (RN)^2 / N_i$$

The signal variance, V_S, of the count rate in the signal-based component of the slope variance is computed as:

$$V_S = \sum_{i=0}^{N-1} K_i^2 \tau_i + \sum_{i<j} 2 K_i K_j \cdot \bar{t}_i$$

Total variance, if desired, is a estimate of the total slope variance V can be computed by adopting $\hat{F}$ as the estimate of the slope:

$$V = V_R + V_S \cdot \hat{F}$$

Exposure-level computations:

The ramps for each resultant are reconstructed from its segments, i, fits by calculating the inverse variance-weighted mean using the read noise variances:

$$w_i = 1 / V_{R_i}$$
$$\hat{F}_{mean} = \frac{\sum_i w_i \hat{F}_i}{\sum_i w_i}$$

The read noise is determined as follows:

$$V_{R_{mean}} = \frac{\sum_i w_i^2 V_{R_i}}{(\sum_i w_i)^2}$$

Finally, the signal variance is calculated as:

$$V_{S_{mean}} = \frac{\sum_i w_i^2 V_{S_i}}{(\sum_i w_i)^2}$$

Upon successful completion of this step, the status attribute ramp_fit will be set to “COMPLETE”.

Jump Detection

For most pixels, the ramp steadily accumulates flux from the sky as an integration proceeds. However, in relatively rare cases, a cosmic ray can pass through the detector which instantaneously deposits a large amount of charge in a pixel. This leads the resulting ramp to have a discontinuous *jump* in a particular read, and accordingly to discontinuities in the resultants downlinked from the telescope. The jump detection algorithm attempts to identify uncontaminated segments of ramps for ramp fitting, so that the underlying astronomical signal can be extracted without contamination from these jumps.

If the uneven-ramp jump detection algorithm is turned on (the default), the ramp fitting algorithm is then run iteratively on a "queue" (list) of ramps. The queue is initialized with the ramp(s). Then following the algorithm presented in Sharma et al (2023) (in preparation), the jump detection algorithm picks a ramp, say $[R_0, \ldots, R_M]$, out of the queue and runs the ramp fitting algorithm on it. It then checks the resulting ramp for jumps. If a jump is detected, then two sub-ramps are created from the passed in ramp which exclude the resultants predicted to be affected by the jump. These sub-ramps are then added to the queue. This process continues until the queue is empty.

Note: It may not always be possible to create two sub-ramps around the resultants predicted to be part of a jump. For example if these jump resultants include the first, second, second-to-last, or last resultant of the ramp then it is not possible to create two meaningful sub-ramps, as one cannot run the ramp fitting algorithm on a ramp with zero or only one resultant. Therefore, in these cases, only one sub-ramp is created and added to the queue.

The method use for determining if and where a jump occurs is divided into two parts. First, a *statistic*, S and possible jump resultants are determined for the fitted ramp. Then the statistic is compared against a threshold function, $S_{\text{threshold}}$ to determine if a jump has occurred.

Statistic and Possible Jump Resultants

The statistic used to determine if a jump has occurred in the ramp, $[R_0, \ldots, R_M]$, is computed from a list of statistics computed for each *single* and *double-difference* of the resultants in the ramp. By single-difference we mean the difference between two adjacent resultants in the ramp, while double-difference refers to the difference between a resultant and a resultant two steps away (the resultant adjacent to a resultant adjacent to the resultant in question).

To compute these statistics, the single-difference excess slope $\delta_{i,i+1}$ and the double-difference excess slope $\delta_{i,i+2}$ are computed as:

$$\delta_{i,i+1} = \frac{R_{i+1} - R_i}{\bar{t}_{i+1} - \bar{t}_i} - \hat{\alpha}$$
$$\delta_{i,i+2} = \frac{R_{i+2} - R_i}{\bar{t}_{i+2} - \bar{t}_i} - \hat{\alpha}$$

where $\hat{\alpha}$ is the slope computed by the ramp fitting algorithm. The variance in the excess slope:

$$Var(\delta_{i,j}) = \frac{Var(R_j - R_i)}{(\bar{t}_j - \bar{t}_i)^2} + f_{corr}(\hat{\alpha})$$
$$Var(R_j - R_i) = \sigma_{RN} \left(\frac{1}{N_j} + \frac{1}{N_i} \right) + \hat{\alpha} \left[\tau_j + \tau_i - \min(\bar{t}_j, \bar{t}_i) \right]$$
$$f_{corr}(\hat{\alpha}) = -\frac{\hat{\alpha}}{t_{M-1} - t_0}$$

where σ_{RN} is the read noise. The single-difference statistic, s'_i, and double-difference statistic, s''_i are,

$$s'_i = \frac{\delta_{i,i+1}}{\sqrt{Var(\delta_{i,i+1})}}$$
$$s''_i = \frac{\delta_{i,i+2}}{\sqrt{Var(\delta_{i,i+2})}}.$$

The statistic s_i for each resultants $0 \leq i \leq M-1$ (no differences from the last resultant are possible) is then computed as:

$$s_i = \begin{cases} s'_i & \text{if } i = M-2 \\ \max(s'_i, s''_i) & \text{otherwise} \end{cases}$$

Finally, $S = \max(s_i)$ is the statistic used to determine if a jump has occurred in the fitted ramp. The possible jump resultants for this ramp are the resultants R_i and R_{i+1}, where $i = \arg\max(s_i)$.

Two possible jump resultants are necessary, because the statistics cannot determine which of the two adjacent resultants is the one affected by the jump. This is because if the jump occurs near the last read making up R_i, then it might appear that R_{i+1} has the jump, this jump will be picked up the s''_i statistic. Using just the s'_i statistic, the jump would be incorrectly identified in R_{i+1}.

Threshold Function

Similarly to the statistic, the threshold function relies on the slope computed by the ramp fitting algorithm, $\hat{\alpha}$. The function itself was determined empirically by running simulations of ramps with jumps and ramps without jumps. The threshold function was determined to be:

$$S_{\text{threshold}}(\hat{\alpha}) = 5.5 - \frac{1}{3}\log_{10}(\hat{\alpha})$$

This corresponds to identifying jumps at 5.5 sigma when the count rate is 1 electron per second, and 4.5 sigma when the count rate is 1000 electrons per second. The decision was made to have the threshold depend on the count rate because the pixels with lots of signal have larger uncertainties; meaning that lower amplitude cosmic rays get identified in these cases.

A jump is determined to have occurred for a ramp fit with statistic, S, with possible jump resultants R_i, R_{i+1}, if $S \geq S_{\text{threshold}}(\hat{\alpha})$. This results in the ramp being split into two sub-ramps $[R_0, \ldots R_{i-1}]$ and $[R_{i+2}, \ldots R_M]$, which are then added to the ramp queue.

Error Propagation

Error propagation in the ramp fitting step is implemented by storing the square-root of the exposure-level combined variance in the ERR array of the primary output product. This combined variance of the exposure-level slope is the sum of the variance of the slope due to the Poisson noise and the variance of the slope due to the read noise. These two variances are also separately written to the arrays VAR_POISSON and VAR_RNOISE in the asdf output.

Arguments

The ramp fitting step has the following optional argument that can be set by the user:

- `--algorithm`: Algorithm to use. Possible values are `ols` and `ols_cas22`. `ols` is the same algorithm used by JWST and can only be used with even ramps. `ols_cas22` needs to be used for uneven ramps. `ols_cas22` is the default.

The following optional arguments are valid only if using the `ols` algorithm.

- `--save_opt`: A True/False value that specifies whether to write the optional output product. Default if False.

- `--opt_name`: A string that can be used to override the default name for the optional output product.
- `--maximum_cores`: The fraction of available cores that will be used for multi-processing in this step. The default value is 'none' which does not use multi-processing. The other options are 'quarter', 'half', and 'all'. Note that these fractions refer to the total available cores and on most CPUs these include physical and virtual cores. The clock time for the step is reduced almost linearly by the number of physical cores used on all machines. For example, on an Intel CPU with six real cores and 6 virtual cores setting maximum_cores to 'half' results in a decrease of a factor of six in the clock time for the step to run. Depending on the system the clock time can also decrease even more with maximum_cores is set to 'all'.
- `--use_ramp_jump_detection`: A True/False value that specifies whether to use the unevenly-spaced jump detection integrated into the ramp fitting algorithm. If True, then the jump detection step will be skipped and then revisited alongside the ramp fitting step. If False, then the jump detection step will be run. The default is True.

Reference Files

The `ramp_fit` step uses two reference file types: GAIN and READNOISE. During ramp fitting, the GAIN values are used to temporarily convert the pixel values from units of DN to electrons, and convert the results of ramp fitting back to DN. The READNOISE values are used as part of the noise estimate for each pixel. Both are necessary for proper computation of noise estimates.

GAIN

READNOISE

Description: Optimized Least-squares with Even Ramps

This step determines the mean count rate, in units of counts per second, for each pixel by performing a linear fit to the data in the input file. The fit is done using the "ordinary least squares" method. The fit is performed independently for each pixel. There can be up to two output files created by the step. The primary output file ("rate") contains the slope at each pixel. A second, optional output product is also available, containing detailed fit information for each pixel. The two types of output files are described in more detail below.

The count rate for each pixel is determined by a linear fit to the cosmic-ray-free and saturation-free ramp intervals for each pixel; hereafter this interval will be referred to as a "segment." The fitting algorithm uses an 'optimal' weighting scheme, as described by Fixsen et al, PASP, 112, 1350. Segments are determined using the 3-D GROUPDQ array of the input data set, under the assumption that the jump step will have already flagged CR's. Segments are terminated where saturation flags are found. Pixels are processed simultaneously in blocks using the array-based functionality of numpy. The size of the block depends on the image size and the number of groups.

The ramp fitting step is also where the *reference pixels* are trimmed, resulting in a smaller array being passed to the subsequent steps.

Multiprocessing

This step has the option of running in multiprocessing mode. In that mode it will split the input data cube into a number of row slices based on the number of available cores on the host computer and the value of the max_cores input parameter. By default the step runs on a single processor. At the other extreme if max_cores is set to 'all', it will use all available cores (real and virtual). Testing has shown a reduction in the elapsed time for the step proportional to the number of real cores used. Using the virtual cores also reduces the elasped time but at a slightly lower rate than the real cores.

Special Cases

If the input dataset has only a single group, the count rate for all unsaturated pixels will be calculated as the value of the science data in that group divided by the group time. If the input dataset has only two groups, the count rate for all unsaturated pixels will be calculated using the differences between the two valid groups of the science data.

For datasets having more than a single group, a ramp having a segment with only a single group is processed differently depending on the number and size of the other segments in the ramp. If a ramp has only one segment and that segment contains a single group, the count rate will be calculated to be the value of the science data in that group divided by the group time. If a ramp has a segment having a single group, and at least one other segment having more than one good group, only data from the segment(s) having more than a single good group will be used to calculate the count rate.

The data are checked for ramps in which there is good data in the first group, but all first differences for the ramp are undefined because the remainder of the groups are either saturated or affected by cosmic rays. For such ramps, the first differences will be set to equal the data in the first group. The first difference is used to estimate the slope of the ramp, as explained in the 'segment-specific computations' section below.

If any input dataset contains ramps saturated in their second group, the count rates for those pixels will be calculated as the value of the science data in the first group divided by the group time.

All Cases

For all input datasets, including the special cases described above, arrays for the primary output (rate) product are computed as follows.

After computing the slopes for all segments for a given pixel, the final slope is determined as a weighted average from all segments, and is written as the primary output product. In this output product, the 3-D GROUPDQ is collapsed into 2-D, merged (using a bitwise OR) with the input 2-D PIXELDQ, and stored as a 2-D DQ array.

A second, 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 3-D arrays called SLOPE, SIGSLOPE, YINT, SIGYINT, WEIGHTS, VAR_POISSON, and VAR_RNOISE that contain the slopes, uncertainties in the slopes, y-intercept, uncertainty in the y-intercept, fitting weights, the variance of the slope due to poisson noise only, and the variance of the slope due to read noise only for each segment of each pixel, respectively. The y-intercept refers to the result of the fit at an effective exposure time of zero. This product also contains a 2-D array called PEDESTAL, which gives the signal at zero exposure time for each pixel, and the 3-D CRMAG array, which contains the magnitude of each group that was flagged as having a CR hit. By default, the name of this output file will have the suffix "_fitopt". In this optional output product, the pedestal array is calculated by extrapolating the final slope (the weighted average of the slopes of all ramp segments) for each pixel from its value at the first group to an exposure time of zero. Any pixel that is saturated on the first group is given a pedestal value of 0. Before compression, the cosmic ray magnitude array is equivalent to the input SCI array but with the only nonzero values being those whose pixel locations are flagged in the input GROUPDQ as cosmic ray hits. The array is compressed, removing all groups in which all the values are 0 for pixels having at least one group with a non-zero magnitude. The order of the cosmic rays within the ramp is preserved.

Slope and Variance Calculations

Slopes and their variances are calculated for each segment, and for the entire exposure. 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, 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', 'o' for segment, or overall (for the entire dataset), respectively.

Optimal Weighting Algorithm

The slope of each segment is calculated using the least-squares method with optimal weighting, as described by Fixsen et al. 2000, PASP, 112, 1350; Regan 2007, JWST-STScI-001212. Optimal weighting determines the relative weighting of each sample when calculating the least-squares fit to the ramp. When the data have low signal-to-noise ratio S, the data are read noise dominated and equal weighting of samples is the best approach. In the high signal-to-noise regime, data are Poisson-noise dominated and the least-squares fit is calculated with the first and last samples. In most practical cases, the data will fall somewhere in between, where the weighting is scaled between the two extremes.

The signal-to-noise ratio S used for weighting selection is calculated from the last sample as:

$$S = \frac{data \times gain}{\sqrt{(read\_noise)^2 + (data \times gain)}},$$

The weighting for a sample i is given as:

$$w_i = (i - i_{midpoint})^P,$$

where $i_{midpoint}$ is the the sample number of the midpoint of the sequence, and P is the exponent applied to weights, determined by the value of S. Fixsen et al. 2000 found that defining a small number of P values to apply to values of S was sufficient; they are given as:

Minimum S	Maximum S	P
0	5	0
5	10	0.4
10	20	1
20	50	3
50	100	6
100		10

Segment-specific Computations:

The variance of the slope of a segment due to read noise is:

$$var_s^R = \frac{12\ R^2}{(ngroups_s^3 - ngroups_s)(group_time^2)},$$

where R is the noise in the difference between 2 frames, $ngroups_s$ is the number of groups in the segment, and $group_time$ is the group time in seconds (from the exposure.group_time).

The variance of the slope in a segment due to Poisson noise is:

$$var_s^P = \frac{slope_{est}}{tgroup \times gain\ (ngroups_s - 1)},$$

where $gain$ is the gain for the pixel (from the GAIN reference file), in e/DN. The $slope_{est}$ is an overall estimated slope of the pixel, calculated by taking the median of the first differences of the groups that are unaffected by saturation and cosmic rays. This is a more robust estimate of the slope than the segment-specific slope, which may be noisy for short segments.

The combined variance of the slope of a segment is the sum of the variances:

$$var_s^C = var_s^R + var_s^P$$

Exposure-level computations:

The variance of the slope due to read noise is:

$$var_o^R = \frac{1}{\sum_s \frac{1}{var_s^R}}$$

where the sum is over all segments.

The variance of the slope due to Poisson noise is:

$$var_o^P = \frac{1}{\sum_s \frac{1}{var_s^P}}$$

The combined variance of the slope is the sum of the variances:

$$var_o^C = var_o^R + var_o^P$$

The square root of the combined variance is stored in the ERR array of the primary output.

The overall slope depends on the slope and the combined variance of the slope of all segments, so is a sum over segments:

$$slope_o = \frac{\sum_s \frac{slope_s}{var_s^C}}{\sum_s \frac{1}{var_s^C}}$$

Upon successful completion of this step, the status attribute ramp_fit will be set to "COMPLETE".

Error Propagation

Error propagation in the ramp fitting step is implemented by storing the square-root of the exposure-level combined variance in the ERR array of the primary output product. This combined variance of the exposure-level slope is the sum of the variance of the slope due to the Poisson noise and the variance of the slope due to the read noise. These two variances are also separately written to the arrays VAR_POISSON and VAR_RNOISE in the asdf output.

For the optional output product, the variance of the slope due to the Poisson noise of the segment-specific slope is written to the VAR_POISSON array. Similarly, the variance of the slope due to the read noise of the segment-specific slope is written to the VAR_RNOISE array.

romancal.ramp_fitting Package

Classes

RampFitStep([name, parent, config_file, ...])	This step fits a straight line to the value of counts vs.

RampFitStep

class romancal.ramp_fitting.**RampFitStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

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, 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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

reference_file_types
spec
weighting

Methods Summary

ols(input_model, readnoise_model, gain_model)	Perform Optimal Linear Fitting on evenly-spaced resultants
ols_cas22(input_model, readnoise_model, ...)	Peform Optimal Linear Fitting on arbitrarily space resulants
process(input)	This is where real work happens.

Attributes Documentation

reference_file_types: ClassVar = ['readnoise', 'gain', 'dark']

spec

```
algorithm = option('ols','ols_cas22', default='ols_cas22')  # Algorithm to use␣
↪to fit.
save_opt = boolean(default=False) # Save optional output
opt_name = string(default='')
```

(continues on next page)

(continued from previous page)

```
maximum_cores = option('none','quarter','half','all',default='none') # max␣
↪number of processes to create
suffix = string(default='rampfit')  # Default suffix of results
use_ramp_jump_detection = boolean(default=True) # Use jump detection during␣
↪ramp fitting
threshold_intercept = float(default=None) # Override the intercept parameter␣
↪for the threshold function in the jump detection algorithm.
threshold_constant = float(default=None) # Override the constant parameter for␣
↪the threshold function in the jump detection algorithm.
```

weighting = 'optimal'

Methods Documentation

ols(*input_model*, *readnoise_model*, *gain_model*)

Perform Optimal Linear Fitting on evenly-spaced resultants

The OLS algorithm used is the same used by JWST for it's ramp fitting.

Parameters

- **input_model** (*RampModel*) – Model containing ramps.
- **readnoise_model** (*ReadnoiseRefModel*) – Model with the read noise reference information.
- **gain_model** (*GainRefModel*) – Model with the gain reference information.

Returns
out_model – Model containing a count-rate image.

Return type
ImageModel

ols_cas22(*input_model*, *readnoise_model*, *gain_model*, *dark_model*)

Peform Optimal Linear Fitting on arbitrarily space resulants

Parameters

- **input_model** (*RampModel*) – Model containing ramps.
- **readnoise_model** (*ReadnoiseRefModel*) – Model with the read noise reference information.
- **gain_model** (*GainRefModel*) – Model with the gain reference information.
- **dark_model** (*DarkRefModel*) – Model with the dark reference information

Returns
out_model – Model containing a count-rate image.

Return type
ImageModel

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

1.4.8 Assign WCS

Description

`romancal.assign_wcs` is the first step run on an image, after `romancal.ramp_fitting`. It associates a World Coordinate System (WCS) object with each science exposure. Note that no fitting is performed in this step; it only creates a WCS object that transforms positions in the `detector` frame to positions in the `world` coordinate frame (ICRS) based on the telescope pointing and reference files in CRDS. The constructed WCS object can be accessed as an attribute of the `meta` object when the 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.

`romancal.assign_wcs` uses GWCS - a package for managing the World Coordinate System of astronomical data. It expects to find a few basic WCS keywords in the `model.meta.wcsinfo` structure. Distortions are stored in reference files in the ASDF format.

`assign_wcs` retrieves reference files from CRDS and creates a pipeline of transforms from input frame `detector` to the telescope frame `v2v3`[1]. 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`.

Note: in earlier builds of the pipeline the distortions are not available.

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

The following attributes in `meta.wcsinfo` are used to define the transform from `v2v3` to `world`:

`RA_REF`, `DEC_REF` - a fiducial point on the sky, ICRS, where the telescope is pointing [deg]

`V2_REF`, `V3_REF` - a point in the V2V3 system which maps to `RA_REF`, `DEC_REF`, [arcsec] This is the reference point of the aperture as defined in the Science Instrument Aperture File (SIAF).

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

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.

[1] V2V3 is a frame defined by the two perpendicular axes that lay along the primary's mirror plane. For completeness, V1 is also part of the telescope frame system, being the axis perpendicular to the primary mirror (i.e. along the telecope's optical axis).

Using the WCS interactively

Once a science 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:

```
>>> from roman_datamodels import datamodels as rdm
>>> image = rdm.open('roman_assign_wcs.asdf')
>>> ra, dec = image.meta.wcs(x, y)
```

The WCS provides access to intermediate coordinate frames and transforms between any two frames in the WCS pipeline in forward or backward direction:

```
>>> image.meta.wcs.available_frames
    ['detector', 'v2v3', 'world']
>>> v2world = image.meta.wcs.get_transform('v2v3', 'world')
>>> ra, dec = v2world(v2, v3)
>>> x1, y1 = image.meta.wcs.invert(ra, dec)
```

There are methods which allow the result of evaluating the WCS object to be an `astropy.SkyCoord` objext (as opposed to numbers) which allows further transformation of coordinates to different coordinate frames.

Simulating a pointing

If one wishes to simulate a pointing on the sky they will need to provide values for the basic WCS keywords. In regular processing these attributes are populated in the Level 1 (raw or `uncal`) files by Science Data Formatting (SDF) using internal databases. The SIAF, in particular, stores information about the apertures including the reference point of each aperture in different coordinate frames associated with the telescope. The following example shows how to get the reference point of an aperture in the V2V3 coordinate system using a package called PySIAF .

```
>>> import pysiaf
>>> siaf = pysiaf.Siaf('Roman')
>>> siaf.apertures # prints the names o fall apertures in the SIAF
>>> ap = siaf['WFI01_FULL']
>>> V2_REF, V3_REF = ap.get_reference_point('tel')
```

romancal.assign_wcs Package

Classes

AssignWcsStep([name, parent, config_file, ...])	Assign a gWCS object to a science image.

AssignWcsStep

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

Bases: *RomanStep*

Assign a gWCS object to a science image.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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: ClassVar = ['distortion']

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

1.4.9 Flatfield Correction

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 bitwise OR operation.

Upon completion of the step, the cal_step attribute "flat_field" gets set to "COMPLETE" in the output science data.

Imaging Data

Imaging data use a straight-forward approach that involves applying a single flat-field reference file to the science image. The processing steps are:

- Find pixels that have a value of NaN or zero in the FLAT reference file SCI array and set their DQ values to "NO_FLAT_FIELD."
- Reset the values of pixels in the flat that have DQ="NO_FLAT_FIELD" to 1.0, so that they have no effect when applied to the science data.
- Apply the flat by dividing it into the science exposure SCI and ERR arrays.
- Propagate the FLAT reference file DQ values into the science exposure DQ array using a bitwise OR operation.

Error Propagation

The VAR_POISSON and VAR_RNOISE variance arrays of the science exposure are divided by the square of the flat-field value for each pixel. A flat-field variance array, VAR_FLAT, is created from the science exposure and flat-field reference file data using the following formula:

The flat-field is applied to the science data, in-place, according to:

$$SCI'_{science} = SCI_{science}/SCI_{flat}$$

The flat-field data is also applied to the VAR_POISSON and VAR_RNOISE variance arrays,

$$VAR\_POISSON_{science} = VAR\_POISSON_{science}/SCI^2_{flat}$$

$$VAR\_RNOISE_{science} = VAR\_RNOISE_{science}/SCI^2_{flat}$$

The variance for the flat-fielded data associated with the science data is determined using,

$$VAR\_FLAT_{science} = ((SCI'_{science})^2/SCI^2_{flat}) * ERR^2_{flat}$$

and finally the error that is associated with the science data is given by,

$$ERR_{science} = \sqrt{VAR\_POISSON + VAR\_RNOISE + VAR\_FLAT}$$

The total ERR array in the science exposure is updated as the square root of the quadratic sum of VAR_POISSON, VAR_RNOISE, and VAR_FLAT.

Reference Files

The `flat_field` step uses a FLAT reference file.

FLAT Reference File

REFTYPE
FLAT

Data model
`roman_datamodels.datamodels.FlatRefModel`

The FLAT reference file contains pixel-by-pixel detector response values. It is used.

Reference Selection Keywords for FLAT

CRDS selects appropriate FLAT references based on the following attributes. FLAT is not applicable for instruments not in the table. Non-standard attributes used for file selection are *required*.

Instrument	Metadata
WFI	instrument, detector, optical_element, date, time

Standard ASDF metadata

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

Attribute	Fully Qualified Path
author	model.meta.author
model_type	model.meta.model_type
date	model.meta.date
description	model.meta.description
instrument	model.meta.instrument.name
reftype	model.meta.reftype
telescope	model.meta.telescope
useafter	model.meta.useafter

NOTE: More information on standard required attributes can be found here: *Standard ASDF metadata*

Type Specific Keywords for FLAT

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

Attribute	Fully qualified path	Instruments
detector	model.meta.instrument.detector	WFI
exptype	model.meta.exposure.type	
optical_element	model.meta.instrument.optical_element	

Reference File Format

FLAT reference files are ASDF format, with 3 data arrays. The format and content of the file is as follows:

Data	Array Type	Dimensions	Data type
data	NDArray	4096 x 4096	float32
err	NDArray	4096 x 4096	float32
dq	NDArray	4096 x 4096	uint32

The ASDF file contains a single set of data, err, and dq arrays.

romancal.flatfield Package

Classes

FlatFieldStep([name, parent, config_file, ...])	Flat-field a science image using a flatfield reference image.

FlatFieldStep

class romancal.flatfield.**FlatFieldStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Flat-field a science image using a flatfield reference image.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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_model)	This is where real work happens.

Attributes Documentation

reference_file_types: ClassVar = ['flat']

Methods Documentation

process(*input_model*)

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

Class Inheritance Diagram

1.4.10 Photometric Calibration

Description

Algorithm

The `photom` step adds flux (photometric) calibrations to the metadata of a data product. The calibration information is read from a photometric reference file, and the exact nature of the calibration information loaded from the reference file is described below. This step does not affect the pixel values of the data product.

Upon successful completion of the photometric correction, the "photom" keyword in "cal_step" in the metadata is set to "COMPLETE".

Photom and Pixel Area Data

The photom reference file contains a table of exposure parameters that define the flux conversion and pixel area data for each optical element. The table contains one row for each optical_element, and the photom step searches the table for the row that matches the parameters of the science exposure and then loads the calibration information from that row of the table.

For these table-based PHOTOM reference files, the calibration information in each row includes a scalar flux conversion constant, the conversion uncertainty, and the nominal pixel area.

The scalar conversion constant is copied to the header keyword "conversion_megajanskys", which gives the conversion from DN/s to megaJy/steradian, and converted to microJy/square arcseconds and saved to the header keyword "conversion_microjanskys". The same process is performed for the uncertainty, with the values saved in "conversion_megajanskys_uncertainty" and "conversion_microjanskys_uncertainty", respectively.

The step also populates the metadata keywords "pixelarea_steradians" and "pixelarea_arcsecsq" in the science data product, which give the average pixel area in units of steradians and square arcseconds, respectively.

Step Arguments

The photometric calibration step has one step-specific argument:

- `--photom`

If the `photom` argument is given with a filename for its value, the photometric calibrated data that are created within the step will be saved to that file.

Reference Files

The `photom` step uses *PHOTOM* reference files.

PHOTOM Reference File

REFTYPE
PHOTOM

Data models
`WfiImgPhotomRefModel`

The PHOTOM reference file contains conversion factors for putting pixel values into physical units.

Reference Selection Keywords for PHOTOM

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

Instrument	Keyword Attributes
WFI	instrument, date, time

Standard ASDF metadata

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

Attribute	Fully Qualified Path
author	model.meta.author
model_type	model.meta.model_type
date	model.meta.date
description	model.meta.description
instrument	model.meta.instrument.name
reftype	model.meta.reftype
telescope	model.meta.telescope
useafter	model.meta.useafter

NOTE: More information on standard required attributes can be found here: *Standard ASDF metadata*

Type Specific Keywords for PHOTOM

In addition to the standard reference file keywords listed above, the following keywords are *required* in PHOTOM reference files. The first (detector) is needed because it is used as a CRDS selector. The second (optical_element) is used to select the appropriate set of photometric parameters. (see *Reference Selection Keywords for PHOTOM*):

Attribute	Fully qualified path	Instruments
detector	model.meta.instrument.detector	WFI
optical_element	model.meta.instrument.optical_element	WFI

Tabular PHOTOM Reference File Format

PHOTOM reference files are ASDF format, with data in the phot_table attribute. The format and content of the file is as follows (see `WfiImgPhotomRefModel`):

Data is stored in a 2D table, with optical elements for the row names:

Instrument	Row names
WFI	F062, F087, F106, F129, F146, F158, F184, F213, GRISM, PRISM, DARK

And the variable attributes for the columns (with data type):

Instrument	Column name	Data type	Dimensions	Units
WFI	photmjsr	quantity	scalar	MJy/steradian
	uncertainty	quantity	scalar	MJy/steradian
	pixelareasr	quantity	scalar	steradian

The pixelareasr variable attribute gives the average pixel area in units of steradians.

romancal.photom Package

Classes

PhotomStep([name, parent, config_file, ...])	PhotomStep: Module for loading photometric conversion information from

PhotomStep

class romancal.photom.**PhotomStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

PhotomStep: Module for loading photometric conversion information from reference files and attaching to the input science data model

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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)	Perform the photometric calibration step

Attributes Documentation

reference_file_types: ClassVar = ['photom']

Methods Documentation

process(*input*)

Perform the photometric calibration step

Parameters
input (*Roman level 2 image datamodel (wfi_image-1.x.x)*) – input roman datamodel

Returns
output_model – output roman datamodel

Return type
Roman level 2 image datamodel (wfi_image-1.x.x)

Class Inheritance Diagram

1.4.11 Flux

Description

Classes
romancal.flux.FluxStep

Alias
flux

This step will apply the flux scale factor, as defined in the meta of the input ImageModel. The step will check units and not re-apply the correction, simply returning the original data.

The `flux` step can take:

- a single 2D input image (in the format of either a string with the full path and filename of an ASDF file or a Roman Datamodel/*ModelContainer*);
- an association table (in JSON format).

This step takes no other parameters than the standard `stpipe` parameters.

The output product are ImageModels, but with the flux scale applied to all the data and variance arrays.

Flux Application

If the input data is in $DN/second$, the flux scale factor, as found in the meta `meta.photometry.conversion_megajanskys`, is simply multiplied to the `data` and `err` arrays. The square of the scale factor is multiplied to all the variance arrays. The resultant units is in MJy/sr.

$$DATA = DATA * SCALE$$
$$ERR = ERR * SCALE$$
$$VAR_{rnoise} = VAR_{rnoise} * SCALE^2$$
$$VAR_{poisson} = VAR_{poisson} * SCALE^2$$
$$VAR_{flat} = VAR_{flat} * SCALE^2$$

Step Arguments

The `flux` step takes no step-specific arguments.

Python Step Interface: FluxStep()

romancal.flux.flux_step Module

Apply the flux scaling

Classes

FluxStep([name, parent, config_file, ...])	Apply flux scaling to count-rate data

FluxStep

class romancal.flux.flux_step.**FluxStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Apply flux scaling to count-rate data

Parameters

input (str, `roman_datamodels.datamodels.DataModel`, or *ModelContainer*) – If a string is provided, it should correspond to either a single ASDF filename or an association filename. Alternatively, a single DataModel instance can be provided instead of an ASDF filename. Multiple files can be processed via either an association file or wrapped by a *ModelContainer*.

Returns

output_models – The models with flux applied.

Return type

`roman_datamodels.datamodels.DataModel`, or *ModelContainer*

Notes

Currently, the correction is done in-place; the inputs are directly modified if in-memory DataModels are input.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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: ClassVar = []

spec

Methods Documentation

process(*input*)

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

Class Inheritance Diagram

romancal.flux Package

Classes

FluxStep([name, parent, config_file, ...])	Apply flux scaling to count-rate data

FluxStep

class romancal.flux.**FluxStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Apply flux scaling to count-rate data

Parameters

input (str, roman_datamodels.datamodels.DataModel, or *ModelContainer*) – If a string is provided, it should correspond to either a single ASDF filename or an association filename. Alternatively, a single DataModel instance can be provided instead of an ASDF filename. Multiple files can be processed via either an association file or wrapped by a *ModelContainer*.

Returns

output_models – The models with flux applied.

Return type

roman_datamodels.datamodels.DataModel, or *ModelContainer*

Notes

Currently, the correction is done in-place; the inputs are directly modified if in-memory DataModels are input.

Create a Step instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – 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: ClassVar = []

spec

Methods Documentation

process(*input*)

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

Class Inheritance Diagram

1.4.12 Source Detection

Description

The source detection step produces catalogs of point-like sources for use by the Tweakreg step for image alignment. It uses DAOStarFinder to detect point sources in the image, with an option to subsequently fit PSF models to the detected sources for more precise centroids and fluxes.

Detecting Sources

Sources are detected using `DAOStarFinder` from photutils, which is an implementation of the method `DAOFIND` from Stetson (1987). The algorithm can be provided limits on the source flux, radius, roundness, sharpness, and background.

PSF Fitting

Star finding algorithms like `DAOStarFinder` provide approximate stellar centroids. More precise centroids may be inferred by fitting model PSFs to the observations. Setting the SourceDetectionStep's option `fit_psf` to True will generate model Roman PSFs with WebbPSF, and fit those models to each of the sources detected by `DAOStarFinder`. More details are in *PSF Fitting*.

Outputs / Returns

By default, the resulting source catalog will be temporarily attached to the output ImageModel in the `meta.source_catalog.tweakreg_catalog` attribute as numpy array representing, in order, source ID, x centroid position, y centroid position, and flux. This catalog will then be deleted from the model in the Tweakreg step.

Optionally, the catalog can be saved to disk in which case a `meta.source_catalog.tweakreg_catalog_name` attribute will be added to the file to point Tweakreg to the catalog on disk. To do this, set `save_catalogs` to True. Output catalogs will be saved in the same directory as input files, and are also 4D numpy arrays representing, in order, source ID, x centroid position, y centroid position, and flux. Output catalogs can be in ASDF or ECSV format.

NOTE: The intermediate resulting ImageModel from SourceDetectionStep can only be saved if it does not contain an attached catalog - to do this, use the `save_catalogs` option to separate the catalog from the file and save them separately.

Options for Thresholding

The DAOStarFinder routine detects point-like sources in an image that are above a certain, specified floating point threshold. This step provides several options for calculating this threshold, either using one value for the entire image, or by detecting sources in segments of the image and using a different appropriate threshold for each (useful if background varies across the image).

The first option is to set `scalar_threshold` - this will use the specified threshold as the detection threshold for the entire image.

The second option is to use `calc_threshold` - this will calculate a single threshold value for the entire image based on the sigma-clipped average (mean, median, or mode) background level of the whole image.

Other Options

Limiting maximum number of sources

By default, all detected sources will be returned in the final output catalog. If you wish to limit the number of sources, this can be done with the `max_sources` argument, which will sort the output catalog by flux and return only the N brightest.

Arguments

The source detection fitting step has several arguments. These can be specified by the user by passing them to the step in a Python session, or setting them in a parameter file.

- **`--kernel_fwhm`: A parameter for DAOStarFinder: size of Gaussian kernel in** pixels. By default the FWHM is assumed to be the diffraction limited PSF, given the filter used for this observation.
- **`--sharplo`: A parameter for DAOStarFinder: lower bound for sharpness.** Default is 0.0.
- **`--sharphi`: A parameter for DAOStarFinder: upper bound for sharpness.** Default is 1.0.
- **`--roundlo`: A parameter for DAOStarFinder: lower bound for roundness.** Default is -1.0. A circular source will have a zero roundness. A source extended in x or y will have a negative or positive roundness, respectively.
- **`--roundhi`: A parameter for DAOStarFinder: upper bound for roundness.** Default is 1.0. A circular source will have a zero roundness. A source extended in x or y will have a negative or positive roundness, respectively.
- **`--peakmax`: A parameter for DAOStarFinder: upper limit on brightest pixel** in sources. Default is 1000.0.
- **`--max_sources`: The maximum number of sources in the output catalog,** choosing brightest. Default is None, which will return all detected sources.
- **`--scalar_threshold`: If specified, the absolute detection threshold to be** used for the entire image. Units are assumed to be the same as input data. One of `scalar_threshold`, `calc_scalar_threshold` must be chosen. Default is None.
- **`--calc_scalar_threshold`: If specified, a single scalar threshold will be** determined for the entire image. This is done by calculating a 2D background image, and using that to determine a single threshold value for the entire image. One of `scalar_threshold` or `calc_scalar_threshold` must be chosen. must be chosen. Default is True.
- **`--snr_threshold`: If using `calc_threshold_img`, the SNR for the threshold** image. Default is 3.0.
- **`--bkg_estimator`: If using `calc_threshold_img`, choice of mean, median, or** mode. Default is median.
- **`--bkg_boxsize`: If using `calc_threshold_img` size of box in pixels for** 2D background / threshold images and if using calc_threshold_2d, the size of the box used when detecting sources. Default is 9.
- **`--bkg_sigma`: If using `calc_threshold_img`, n sigma for sigma clipping** for background calculation. Default is 2.0.
- **`--bkg_filter_size`: If using `calc_threshold_img` or `calc_threshold_2d`,** size of square gaussian kernel for background calculation. Default is 3.
- **`--save_catalogs`: A True/False value that specifies whether to write** the optional output catalog. Default is False.
- **`--output_cat_filetype`: If `save_catalogs` is True, file type of output** catalog from choice of asdf and escv. Catalog will be saved as a numpy array with four dimensions. In order, these represent source ID, x centroid position, y centroid position, and flux.

- **`--fit_psf`: If True, fit a PSF model to each detected source for more precise** source centroids and fluxes.

PSF Fitting

A few PSF fitting utilities are included to interface between observations within Roman datamodels and methods within dependencies that generate and fit PSF models to observations.

Create PSF models

create_gridded_psf_model computes a gridded PSF model for a given detector using `CreatePSFLibrary` from WebbPSF. The defaults are chosen to balance more accurate PSF models with the cost of increased runtime. For further reading on the WebbPSF approach to ePSFs, see the WebbPSF docs on Using PSF Grids.

Fit model PSFs to an ImageModel

Once PSF models are generated, you can fit those models to observations in an ImageModel with *fit_psf_to_image_model* using photutils. By default the fits are done with `PSFPhotometry`, and crowded fields may benefit from using `IterativePSFPhotometry`. For neighboring sources that are near one another on the detector, grouping the sources and fitting their PSFs simultaneously improves the fit quality. Initial guesses for target centroids can be given or source detection can be performed with, e.g., `DAOStarFinder`.

romancal.source_detection Package

Classes

SourceDetectionStep([name, parent, ...])	SourceDetectionStep: Detect point-like sources in image to create a catalog for alignment in tweakreg.

SourceDetectionStep

class romancal.source_detection.**SourceDetectionStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

SourceDetectionStep: Detect point-like sources in image to create a catalog for alignment in tweakreg.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.

- ****kws** (*dict*) – 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

```
kernel_fwhm = float(default=None)  # DAOStarFinder:Size of Gaussian kernel,
# in pixels.
sharplo = float(default=0.)  # DAOStarFinder: Lower bound for sharpness.
# Typical values of sharpness range from 0 (flat) to 1 (delta function).
sharphi = float(default=1.0)  # DAOStarFinder: Upper bound for sharpness.
roundlo = float(default=-1.0)  # DAOStarFinder: Lower bound for roundness.
# A circular source will have a zero roundness. A source extended in x or
# y will have a negative or positive roundness, respectively.
roundhi = float(default=1.0)  # DAOStarFinder: Upper bound for roundness.
peakmax = float(default=100000.0)  # Upper limit on brightest pixel in sources.
max_sources = float(default=None)  # Max number of sources, choosing brightest.
scalar_threshold = float(default=None) # Detection threshold, to
# be used for entire image. Assumed to be in same units as data, and is
# an absolute threshold over background.
calc_threshold = boolean(default=True) # Calculate a single absolute
# detection threshold from image based on background.
snr_threshold = float(default=3.0)  # if calc_threshold_img,
# the SNR for the threshold image.
bkg_estimator = string(default='median')  # if calc_threshold_img,
# choice of mean, median, or mode.
bkg_boxsize = integer(default=9)  # if calc_threshold_img,
# size of box in pixels for 2D background.
bkg_sigma = float(default=2.0) # if calc_threshold_img,
# n sigma for sigma clipping bkgrnd.
bkg_filter_size = integer(default=3) # if calc_threshold_img,
# size of Gaussian kernel for background.
save_catalogs = boolean(default=False) # Save source catalog to file?
# Will overwrite an existing catalog of the same name.
output_cat_filetype = option('asdf', 'ecsv', default='asdf') # Used if
#save_catalogs=True - file type of output catalog.
fit_psf = boolean(default=False)  # fit source PSFs for accurate astrometry
```

(continues on next page)

(continued from previous page)

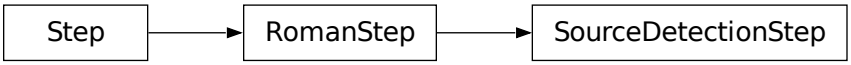
```
min_separation = integer(default=11)  # don't find multiple sources closer
# than this number of pixels
```

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



romancal.lib.psf Module

Utilities for fitting model PSFs to rate images.

Functions

create_gridded_psf_model(path_prefix, filt, ...)	Compute a gridded PSF model for one SCA via `CreatePSFLibrary`.
fit_psf_to_image_model([image_model, data, ...])	Fit PSF models to an `ImageModel`.
dq_to_boolean_mask(image_model_or_dq[, ...])	Convert a DQ bitmask to a boolean mask.

create_gridded_psf_model

romancal.lib.psf.**create_gridded_psf_model**(*path_prefix*, *filt*, *detector*, *oversample=11*, *fov_pixels=9*, *sqrt_n_psfs=2*, *overwrite=False*, *buffer_pixels=100*, *instrument_options=None*, *logging_level=None*)

Compute a gridded PSF model for one SCA via `CreatePSFLibrary`.

Parameters

- **path_prefix** (*str or Path-like*) – Prefix to the output file path for the gridded PSF model FITS file. A suffix denoting the detector name will be appended.
- **filt** (*str*) – Filter name, starting with "F". For example: `"F184"`.
- **detector** (*str*) – Computed gridded PSF model for this SCA. Examples include: `"SCA01"` or `"SCA18"`.

- **oversample** (*int, optional*) – Oversample factor, default is 11. See WebbPSF docs for details[1]. Choosing an odd number makes the pixel convolution more accurate.
- **fov_pixels** (*int, optional*) – Field of view width [pixels]. Default is 12. See WebbPSF docs for details[1].
- **sqrt_n_psfs** (*int, optional*) – Square root of the number of PSFs to calculate, distributed uniformly across the detector. Default is 4.
- **overwrite** (*bool, optional*) – Overwrite output file if one already exists. Default is False.
- **buffer_pixels** (*int, optional*) – Calculate a grid of PSFs distributed uniformly across the detector at least `buffer_pixels` away from the detector edges. Default is 100.
- **instrument_options** (*dict, optional*) – Instrument configuration options passed to WebbPSF. For example, WebbPSF assumes Roman pointing jitter consistent with mission specs by default, but this can be turned off with: `{'jitter': None, 'jitter_sigma': 0}`.
- **logging_level** (*str, optional*) – Set logging level by name if not `None`, otherwise inherit from the romancal logger.

Returns

- **gridmodel** (`photutils.psf.GriddedPSFModel`) – Gridded PSF model evaluated at several locations on one SCA.
- **model_psf_centroids** (*list of tuples*) – Pixel locations of the PSF models calculated for `gridmodel`.

References

fit_psf_to_image_model

romancal.lib.psf.**fit_psf_to_image_model**(*image_model=None*, *data=None*, *error=None*, *dq=None*, *photometry_cls=<class 'photutils.psf.photometry.PSFPhotometry'>*, *psf_model=None*, *grouper=None*, *fitter=None*, *localbkg_estimator=None*, *finder=None*, *x_init=None*, *y_init=None*, *progress_bar=False*, *error_lower_limit=None*, *fit_shape=(15, 15)*, *exclude_out_of_bounds=True*)

Fit PSF models to an `ImageModel`.

Parameters

- **image_model** (`roman_datamodels.datamodels.ImageModel`) – Image datamodel. If `image_model` is supplied, `data,error,dq` should be `None`.
- **data** (`astropy.units.Quantity`) – Fit a PSF model to the rate image `data`. If `data, error,dq` are supplied, `image_model` should be `None`.
- **error** (`astropy.units.Quantity`) – Uncertainties on fluxes in `data`. Should be `None` if `image_model` is supplied.
- **dq** (`numpy.ndarray`) – Data quality bitmask for `data`. Should be `None` if `image_model` is supplied.

[1] `WebbPSF documentation for `webbpsf.JWInstrument.calc_psf <https://webbpsf.readthedocs.io/en/latest/api/webbpsf.JWInstrument.html#webbpsf.JWInstrument.calc_psf>`_`

- **photometry_cls** ({photutils.psf.PSFPhotometry,} – photutils.psf.IterativePSFPhotometry} Choose a photutils PSF photometry technique (default or iterative).
- **psf_model** (astropy.modeling.Fittable2DModel) – The 2D PSF model to fit to the rate image. Usually this model is an instance of photutils.psf.GriddedPSFModel.
- **grouper** (photutils.psf.SourceGrouper) –

  Specifies rules for attempting joint fits of multiple PSFs when
  there are nearby sources at small separations.
- **fitter** (astropy.modeling.fitting.Fitter, optional) – Modeling class which optimizes the PSF fit. Default is astropy.modeling.fitting.LevMarLSQFitter(calc_uncertainties=True).
- **localbkg_estimator** (photutils.background.LocalBackground, optional) – Specifies inner and outer radii for computing flux background near a source. Default has inner_radius=10, outer_radius=30.
- **finder** (subclass of photutils.detection.StarFinderBase, optional) – When photutils_cls is photutils.psf.IterativePSFPhotometry, the finder is called to determine if sources remain in the rate image after one PSF model is fit to the observations and removed. Default was extracted from the DAOStarFinder call in the Source Detection step.
- **x_init** (numpy.ndarray, optional) – Initial guesses for the x pixel coordinates of each source to fit.
- **y_init** (numpy.ndarray, optional) – Initial guesses for the y pixel coordinates of each source to fit.
- **progress_bar** (*bool, optional*) – Render a progress bar via photutils. Default is False.
- **error_lower_limit** (astropy.units.Quantity, optional) – Since some synthetic images may have bright sources with very small statistical uncertainties, the error can be clipped at error_lower_limit to prevent over-confident fits.
- **fit_shape** (*int, or tuple of length 2, optional*) – Rectangular shape around the center of a star that will be used to define the PSF-fitting data. See docs for photutils.psf.PSFPhotometry for details. Default is (16, 16).
- **exclude_out_of_bounds** (*bool, optional*) – If True, do not attempt to fit stars which have initial centroids that fall outside the pixel limits of the SCA. Default is False.

Returns

- **results_table** (astropy.table.QTable) – PSF photometry results.
- **photometry** (instance of class photutils_cls) – PSF photometry instance with configuration settings and results.

dq_to_boolean_mask

romancal.lib.psf.**dq_to_boolean_mask**(*image_model_or_dq*, *ignore_flags=0*, *flag_map_name='ROMAN_DQ'*)

Convert a DQ bitmask to a boolean mask. Useful for photutils methods.

Parameters

- **image_model_or_dq** (roman_datamodels.datamodels.ImageModel or numpy.ndarray) – ImageModel containing the DQ bitmask to convert to a boolean mask, or the DQ bitmask itself.
- **ignore_flags** (*int, str, list, None (default = 0)*) – See docs for astropy.nddata.bitmask.extend_bit_flag_map
- **flag_map_name** (*str*) – Name for the bitmask flag map in the astropy bitmask registry

Returns

mask – Boolean mask

Return type

numpy.ndarray

1.4.13 TweakReg

Description

Class

roman.tweakreg.TweakRegStep

Alias

tweakreg

Overview

This step uses the coordinates of point-like sources from an input catalog (i.e. the result from SourceDetectionStep saved in the meta.tweakreg_catalog attribute) and compares them with the coordinates from a Gaia catalog to compute corrections to the WCS of the input images such that sky catalogs obtained from the image catalogs using the corrected WCS will align on the sky.

Custom Source Catalogs

The default catalog used by tweakreg step can be disabled by providing a file name to a custom source catalog in the meta.tweakreg_catalog attribute of input data models. The catalog must be in a format automatically recognized by read(). The catalog must contain either 'x' and 'y' or 'xcentroid' and 'ycentroid' columns which indicate source *image* coordinates (in pixels). Pixel coordinates are 0-indexed.

For the tweakreg step to use user-provided input source catalogs, use_custom_catalogs parameter of the tweakreg step must be set to True.

In addition to setting the meta.tweakreg_catalog attribute of input data models to the custom catalog file name, the tweakreg_step also supports two other ways of supplying custom source catalogs to the step:

1. Adding tweakreg_catalog attribute to the members of the input ASN table - see ModelContainer for more details. Catalog file names are relative to ASN file path.

2. Providing a simple two-column text file, specified via step's parameter `catfile`, that contains input data models' file names in the first column and the file names of the corresponding catalogs in the second column. Catalog file names are relative to `catfile` file path.

Specifying custom source catalogs via either the input ASN table or `catfile`, will update input data models' `meta.tweakreg_catalog` attributes to the catalog file names provided in either in the ASN table or `catfile`.

Note: When custom source catalogs are provided via both `catfile` and ASN table members' attributes, the `catfile` takes precedence and catalogs specified via ASN table are ignored altogether.

Note:

1. Providing a data model file name in the `catfile` and leaving the corresponding source catalog file name empty – same as setting `'tweakreg_catalog'` in the ASN table to an empty string `""` – would set corresponding input data model's `meta.tweakreg_catalog` attribute to `None`. In this case, `tweakreg_step` will automatically generate a source catalog for that data model.
2. If an input data model is not listed in the `catfile` or does not have `'tweakreg_catalog'` attribute provided in the ASN table, then the catalog file name in that model's `meta.tweakreg_catalog` attribute will be used. If `model.meta.tweakreg_catalog` is `None`, `tweakreg_step` will automatically generate a source catalog for that data model.

Alignment

The source catalog (either created by `SourceDetectionStep` or provided by the user) gets cross-matched and fit to an astrometric reference catalog (set by `TweakRegStep.abs_refcat`) and the results are stored in `model.meta.wcs_fit_results`. The pipeline initially supports fitting to any Gaia Data Release (defaults to `GAIADR3`).

An example of the content of `model.meta.wcs_fit_results` is as follows:

```
model.meta.wcs_fit_results = {
  "status": "SUCCESS",
  "fitgeom": "rshift",
  "matrix": array([[ 1.00000000e+00,  1.04301609e-13],
        [-1.04301609e-13,  1.00000000e+00]]),
  "shift": array([ 7.45523163e-11, -1.42718944e-10]),
  "center": array([-183.87997841, -119.38467775]),
  "proper_rot": 5.9760419875149846e-12,
  "proper": True,
  "rot": (5.9760419875149846e-12, 5.9760419875149846e-12),
  "<rot>": 5.9760419875149846e-12,
  "scale": (1.0, 1.0),
  "<scale>": 1.0,
  "skew": 0.0,
  "rmse": 2.854152848489525e-10,
  "mae": 2.3250544963289652e-10,
  "nmatches": 22
}
```

Details about most of the parameters available in `model.meta.wcs_fit_results` can be found on the TweakWCS's webpage, under its linearfit module.

WCS Correction

The linear coordinate transformation computed in the previous step is used to define tangent-plane corrections that need to be applied to the GWCS pipeline in order to correct input image WCS. This correction is implemented by inserting a `v2v3corr` frame with tangent plane corrections into the GWCS pipeline of the image's WCS.

Step Arguments

`TweakRegStep` has the following arguments:

Catalog parameters:

- `use_custom_catalogs`: A boolean that indicates whether to ignore source catalog in the input data model's `meta.tweakreg_catalog` attribute (Default=`False`).

  Note: If `True`, the user must provide a valid custom catalog that will be assigned to `meta.tweakreg_catalog` and used throughout the step.

- `catalog_format`: A `str` indicating one of the catalog output file format supported by `astropy.table.Table` (Default='ascii.ecsv').

  Note:

  - This option must be provided whenever `use_custom_catalogs = True`.
  - The full list of supported formats can be found on the `astropy`'s Built-In Table Readers/Writers webpage.

- `catfile`: Name of the file with a list of custom user-provided catalogs (Default='').

  Note:

  - This option must be provided whenever `use_custom_catalogs = True`.

- `catalog_path`: A `str` indicating the catalogs output file path (Default='').

  Note: All catalogs will be saved to this path. The default value is the current working directory.

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` indicating minimum number of objects acceptable for matching (Default=15).
- `searchrad`: A `float` indicating the search radius in arcsec for a match (Default=2.0).
- `use2dhist`: A boolean indicating whether to use 2D histogram to find initial offset (Default=True).
- `separation`: Minimum object separation in arcsec (Default=1.0).
- `tolerance`: Matching tolerance for `xyxymatch` in arcsec (Default=0.7).

Catalog fitting parameters:

- `fitgeometry`: A `str` value indicating the type of affine transformation to be considered when fitting catalogs. Allowed values:
  - `'shift'`: x/y shifts only
  - `'rshift'`: rotation and shifts
  - `'rscale'`: rotation and scale
  - `'general'`: shift, rotation, and scale

  The default value is "rshift".

Note: Mathematically, alignment of images observed in different tangent planes requires `fitgeometry='general'` in order to fit source catalogs in the different images even if mis-alignment is caused only by a shift or rotation in the tangent plane of one of the images.

However, under certain circumstances, such as small alignment errors or minimal dithering during observations that keep tangent planes of the images to be aligned almost parallel, then it may be more robust to use a `fitgeometry` setting with fewer degrees of freedom such as `'rshift'`, especially for "ill-conditioned" source catalogs such as catalogs with very few sources, or large errors in source positions, or sources placed along a line or bunched in a corner of the image (not spread across/covering the entire image).

- `nclip`: A non-negative integer number of clipping iterations to use in the fit (Default = 3).
- `sigma`: A positive `float` indicating the clipping limit, in sigma units, used when performing fit (Default=3.0).

Absolute Astrometric fitting parameters:

Parameters used for absolute astrometry to a reference catalog.

- `abs_refcat`: String indicating what astrometric catalog should be used. Currently supported options are (Default='GAIADR3'): `'GAIADR1'`, `'GAIADR2'`, or `'GAIADR3'`.

Note: If `None` or an empty string is passed in, `TweakRegStep` will use the default catalog as set by `tweakreg_step.DEFAULT_ABS_REFCAT`.

- `abs_minobj`: A positive `int` indicating minimum number of objects acceptable for matching. (Default=15).
- `abs_searchrad`: A `float` indicating the search radius in arcsec for a match. It is recommended that a value larger than `searchrad` be used for this parameter (e.g. 3 times larger) (Default=6.0).
- `abs_use2dhist`: A boolean indicating whether to use 2D histogram to find initial offset. It is strongly recommended setting this parameter to `True`. Otherwise the initial guess for the offsets will be set to zero (Default=True).
- `abs_separation`: Minimum object separation in arcsec. It is recommended that a value smaller than `separation` be used for this parameter (e.g. 10 times smaller) (Default=0.1).
- `abs_tolerance`: Matching tolerance for `xyxymatch` in arcsec (Default=0.7).
- `abs_fitgeometry`: A `str` value indicating the type of affine transformation to be considered when fitting catalogs. Allowed values:
  - `'shift'`: x/y shifts only
  - `'rshift'`: rotation and shifts
  - `'rscale'`: rotation and scale
  - `'general'`: shift, rotation, and scale

The default value is "rshift". Note that the same conditions/restrictions that apply to `fitgeometry` also apply to `abs_fitgeometry`.

- `abs_nclip`: A non-negative integer number of clipping iterations to use in the fit (Default = 3).
- `abs_sigma`: A positive `float` indicating the clipping limit, in sigma units, used when performing fit (Default=3.0).
- `save_abs_catalog`: A boolean specifying whether or not to write out the astrometric catalog used for the fit as a separate product (Default=False).

Further Documentation

The underlying algorithms as well as formats of source catalogs are described in more detail on the TweakWCS webpage.

Examples

In the examples below, `img` is either a string with the filename of a Roman `ASDF` file or a Roman datamodel `ImageModel`.

1. To run TweakReg in a Python session with the default parameters:

```
from romancal.tweakreg.tweakreg_step import TweakRegStep
step = TweakRegStep()
step.call([img])
```

Note: If the input is a single Roman `DataModel`, either `step.call([img])` or `step.call(img)` will work. For multiple elements as input, they must be passed in as either a list or a ModelContainer.

2. To run TweakReg in a Python session on an association file with the default parameters:

```
from romancal.tweakreg.tweakreg_step import TweakRegStep
step = TweakRegStep()
step.call("asn_file.json")
```

3. To run TweakReg on a Roman's exposure with default astrometric parameters and save the absolute catalog data:

```
from romancal.tweakreg.tweakreg_step import TweakRegStep
step = TweakRegStep()
step.save_abs_catalog = True # save the catalog data used for absolute␣
↪astrometry
step.abs_refcat = 'GAIADR3' # use Gaia DR3 for absolute astrometry
step.catalog_path = '/path/for/the/abs/catalog' # save the Gaia catalog to␣
↪this path
step.call([img])
```

4. To run TweakReg using a custom source catalog with the default parameters:

   - make sure the source catalog is in one of the supported formats. The file content below is an example of a valid catalog (`ascii.csv` format):

```
$ cat ref_catalog_1
```

(continues on next page)

(continued from previous page)

```
x,y
846.1321662446178,945.839358133909
664.7073537074112,1028.4613139252003
1036.160742774408,642.3379043578552
935.8827367579428,594.1745467413945
1491.9672737821606,1037.4723609624757
735.1256651803337,1410.2791591559157
1358.2876707625007,651.7112260833995
526.4715950130742,751.745104066621
1545.082698426152,703.601696337681
374.9609365496525,972.6561578187437
1110.3498547121228,1644.2214966576498
341.18333252240654,891.4733849441861
820.0520846885105,312.0088351823117
567.7054174813052,386.8883078361564
1447.356249085851,1620.3390168916592
1400.4271386280673,1674.3765672924937
1681.9744852889235,571.6748779060324
959.7317254404431,197.8757865066898
1806.3360866990297,769.0603031839573
487.1560001146406,257.30706691141086
1048.7910126076483,85.36675265982751
1075.508595999755,29.085099663125334
```

- create `catfile` containing the filename of the input Roman datamodel and its corresponding catalog, one per line, as shown below

```
$ cat /path/to/catfile/catfilename

img1 ref_catalog_1
img2 ref_catalog_2
img3 ref_catalog_3
```

The content of `catfile` will allow TweakReg to assign the custom catalog to the correct input Roman datamodel. In the example above, source catalog `ref_catalog_1` will be assign to `img1`, and so on.

Now we can execute the following:

```
from romancal.tweakreg.tweakreg_step import TweakRegStep
step = TweakRegStep()
step.use_custom_catalogs = True # use custom catalogs
step.catalog_format = "ascii.ecsv" # custom catalogs format
step.catfile = '/path/to/catfile/catfilename' # path to datamodel:catalog␣
↪mapping
step.call([img])
```

Also See:

tweakreg_step

The tweakreg_step function (class name TweakRegStep) is the top-level function used to call the "tweakreg" operation from the Roman Calibration pipeline. Roman pipeline step for image alignment.

class romancal.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, 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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

class_alias = 'tweakreg'

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.

refcat = None

reference_file_types: ClassVar = []

```
spec = '\n use_custom_catalogs = boolean(default=False) # Use custom user-provided
catalogs?\n catalog_format = string(default=\'ascii.ecsv\') # Catalog output file
format\n catfile = string(default=\'\') # Name of the file with a list of custom
user-provided catalogs\n catalog_path = string(default=\'\') # Catalog output file
path\n enforce_user_order = boolean(default=False) # Align images in user specified
order?\n expand_refcat = boolean(default=False) # Expand reference catalog with new
sources?\n minobj = integer(default=15) # Minimum number of objects acceptable for
matching\n searchrad = float(default=2.0) # The search radius in arcsec for a
match\n use2dhist = boolean(default=True) # Use 2d histogram to find initial
offset?\n separation = float(default=1.0) # Minimum object separation in arcsec\n
tolerance = float(default=0.7) # Matching tolerance for xyxymatch in arcsec\n
fitgeometry = option(\'shift\', \'rshift\', \'rscale\', \'general\',
default=\'rshift\') # Fitting geometry\n nclip = integer(min=0, default=3) # Number
of clipping iterations in fit\n sigma = float(min=0.0, default=3.0) # Clipping limit
in sigma units\n abs_refcat = string(default=\'GAIADR3\') # Absolute reference\n #
catalog. Options:  "\'GAIADR3\'", "\'GAIADR2\'", or "\'GAIADR1\'"\n save_abs_catalog
= boolean(default=False) # Write out used absolute astrometric reference catalog as
a separate product\n abs_minobj = integer(default=15) # Minimum number of objects
acceptable for matching when performing absolute astrometry\n abs_searchrad =
float(default=6.0) # The search radius in arcsec for a match when performing
absolute astrometry\n # We encourage setting this parameter to True. Otherwise,
xoffset and yoffset will be set to zero.\n abs_use2dhist = boolean(default=True) #
Use 2D histogram to find initial offset when performing absolute astrometry?\n
abs_separation = float(default=0.1) # Minimum object separation in arcsec when
performing absolute astrometry\n abs_tolerance = float(default=0.7) # Matching
tolerance for xyxymatch in arcsec when performing absolute astrometry\n # Fitting
geometry when performing absolute astrometry\n abs_fitgeometry = option(\'shift\',
\'rshift\', \'rscale\', \'general\', default=\'rshift\')\n abs_nclip =
integer(min=0, default=3) # Number of clipping iterations in fit when performing
absolute astrometry\n abs_sigma = float(min=0.0, default=3.0) # Clipping limit in
sigma units when performing absolute astrometry\n output_use_model =
boolean(default=True) # When saving use `DataModel.meta.filename`\n '
```

astrometric_utils

The `astrometric_utils` module provides functions for generating astrometric catalogs of sources for the field-of-view covered by a set of images.

`romancal.tweakreg.astrometric_utils.`**compute_radius**(*wcs*)

Compute the radius from the center to the furthest edge of the WCS.

`romancal.tweakreg.astrometric_utils.`**create_astrometric_catalog**(*input_models*, *catalog='GAIADR3'*, *output='ref_cat.ecsv'*, *gaia_only=False*, *table_format='ascii.ecsv'*, *existing_wcs=None*, *num_sources=None*, *epoch=None*)

Create an astrometric catalog that covers the inputs' field-of-view.

Parameters

- **input_models** (*str, list*) – Filenames of images to be aligned to astrometric catalog

- **catalog** (*str, optional*) – Name of catalog to extract astrometric positions for sources in the input images' field-of-view. Default: GAIADR3. Options available are documented on the catalog web page.
- **output** (*str, optional*) – Filename to give to the astrometric catalog read in from the master catalog web service. If None, no file will be written out.
- **gaia_only** (*bool, optional*) – Specify whether or not to only use sources from GAIA in output catalog
- **table_format** (*str, optional*) – Format to be used when writing the results to a file using the `output` option. A full list of the options can be found here: https://docs.astropy.org/en/stable/io/unified.html#built-in-readers-writers
- **existing_wcs** (*model*) – existing WCS object specified by the user as generated by `resample.resample_utils.make_output_wcs`
- **num_sources** (*int*) – Maximum number of brightest/faintest sources to return in catalog. If `num_sources` is negative, return that number of the faintest sources. By default, all sources are returned.
- **epoch** (*float or str, optional*) – Reference epoch used to update the coordinates for proper motion (in decimal year). If `None` then the epoch is obtained from the metadata.

Notes

This function will point to astrometric catalog web service defined through the use of the `ASTROMETRIC_CATALOG_URL` environment variable. Also, the default catalog to be used is set by the `DEF_CAT` variable.

Returns
ref_table – Astropy Table object of the catalog

Return type
`Table`

`romancal.tweakreg.astrometric_utils.`**`get_catalog`**(*ra*, *dec*, *epoch=2016.0*, *sr=0.1*, *catalog='GAIADR3'*, *timeout=30.0*)

Extract catalog from VO web service.

Parameters

- **ra** (*float*) – Right Ascension (RA) of center of field-of-view (in decimal degrees)
- **dec** (*float*) – Declination (Dec) of center of field-of-view (in decimal degrees)
- **epoch** (*float, optional*) – Reference epoch used to update the coordinates for proper motion (in decimal year). Default: 2016.0.
- **sr** (*float, optional*) – Search radius (in decimal degrees) from field-of-view center to use for sources from catalog. Default: 0.1 degrees
- **catalog** (*str, optional*) – Name of catalog to query, as defined by web-service. Default: 'GAIADR3'
- **timeout** (*float, optional*) – Set the request timeout (in seconds). Default: 30 s.

Return type
A Table object of returned sources with all columns as provided by catalog.

romancal.tweakreg Package

Classes

TweakRegStep([name, parent, config_file, ...])	TweakRegStep: Image alignment based on catalogs of sources detected in input images.

TweakRegStep

class romancal.tweakreg.**TweakRegStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

TweakRegStep: Image alignment based on catalogs of sources detected in input images.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
refcat
reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'tweakreg'

refcat = None

reference_file_types: ClassVar = []

spec

```
use_custom_catalogs = boolean(default=False) # Use custom user-provided␣
↪catalogs?
catalog_format = string(default='ascii.ecsv') # Catalog output file format
catfile = string(default='') # Name of the file with a list of custom user-
↪provided catalogs
catalog_path = string(default='') # Catalog output file path
enforce_user_order = boolean(default=False) # Align images in user specified␣
↪order?
expand_refcat = boolean(default=False) # Expand reference catalog with new␣
↪sources?
minobj = integer(default=15) # Minimum number of objects acceptable for matching
searchrad = float(default=2.0) # The search radius in arcsec for a match
use2dhist = boolean(default=True) # Use 2d histogram to find initial offset?
separation = float(default=1.0) # Minimum object separation in arcsec
tolerance = float(default=0.7) # Matching tolerance for xyxymatch in arcsec
fitgeometry = option('shift', 'rshift', 'rscale', 'general', default='rshift')
↪# Fitting geometry
nclip = integer(min=0, default=3) # Number of clipping iterations in fit
sigma = float(min=0.0, default=3.0) # Clipping limit in sigma units
abs_refcat = string(default='GAIADR3')  # Absolute reference
# catalog. Options: "'GAIADR3'", "'GAIADR2'", or "'GAIADR1'"
save_abs_catalog = boolean(default=False)  # Write out used absolute␣
↪astrometric reference catalog as a separate product
abs_minobj = integer(default=15) # Minimum number of objects acceptable for␣
↪matching when performing absolute astrometry
abs_searchrad = float(default=6.0) # The search radius in arcsec for a match␣
↪when performing absolute astrometry
# We encourage setting this parameter to True. Otherwise, xoffset and yoffset␣
↪will be set to zero.
abs_use2dhist = boolean(default=True) # Use 2D histogram to find initial offset␣
↪when performing absolute astrometry?
abs_separation = float(default=0.1) # Minimum object separation in arcsec when␣
↪performing absolute astrometry
abs_tolerance = float(default=0.7) # Matching tolerance for xyxymatch in arcsec␣
↪when performing absolute astrometry
# Fitting geometry when performing absolute astrometry
abs_fitgeometry = option('shift', 'rshift', 'rscale', 'general', default='rshift
↪')
```

(continues on next page)

(continued from previous page)

```
abs_nclip = integer(min=0, default=3) # Number of clipping iterations in fit␣
↪when performing absolute astrometry
abs_sigma = float(min=0.0, default=3.0) # Clipping limit in sigma units when␣
↪performing absolute astrometry
output_use_model = boolean(default=True)  # When saving use `DataModel.meta.
↪filename`
```

Methods Documentation

process(*input*)

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

Class Inheritance Diagram

1.4.14 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 doing cosmic-ray rejection and combining multiple images into a mosaic. When running the `skymatch` step in a matching mode, it compares *total* signal levels in *the overlap regions* of a set of input images and computes the signal offsets for each image that will minimize – in a least squares sense – the residuals across the entire set. This comparison is performed directly on the input images without resampling them onto a common grid. The overlap regions are computed directly on the sky (celestial sphere) for each pair of input images. Matching based on total signal level is especially useful for images that are dominated by large, diffuse sources, where it is difficult – if not impossible – to find and measure true sky.

Note that the meaning of "sky background" depends on the chosen sky computation method. When the matching method is used, for example, the reported "sky" value is only the offset in levels between images and does not necessarily include the true total sky level.

Note: Throughout this document the term "sky" is used in a generic sense, referring to any kind of non-source background signal, which may include actual sky, as well as instrumental (e.g. thermal) background, etc.

The step records information in three attributes that are included in the output files:

- method: records the sky method that was used to compute sky levels
- level: the sky level computed for each image
- subtract: a boolean indicating whether or not the sky was subtracted from the output images. Note that by default the step argument "subtract" is set to `False`, which means that the sky will *NOT* be subtracted (see the *skymatch step arguments* for more details).

Both the "subtract" and "level" attributes are important information for downstream tasks, such as outlier detection and resampling. Outlier detection will use the "level" values to internally equalize the images, which is necessary to prevent false detections due to overall differences in signal levels between images, and the resample step will subtract the "level" values from each input image when combining them into a mosaic.

Assumptions

When matching sky background, the code needs to compute bounding polygon intersections in world coordinates. The input images, therefore, need to have a valid WCS, generated by the *assign_wcs* step.

Algorithms

The `skymatch` step provides several methods for constant sky background value computations.

The first method, called "local", essentially is an enhanced version of the original sky subtraction method used in older versions of astrodrizzle. This method simply computes the mean/median/mode/etc. value of the sky separately in each input image. This method was upgraded to be able to use DQ flags to remove bad pixels from being used in the computations of sky statistics.

In addition to the classic "local" method, two other methods have been introduced: "global" and "match", as well as a combination of the two – "global+match".

- The "global" method essentially uses the "local" method to first compute a sky value for each image separately, and then assigns the minimum of those results to all images in the collection. Hence after subtraction of the sky values only one image will have a net sky of zero, while the remaining images will have some small positive residual.
- The "match" algorithm computes only a correction value for each image, such that, when applied to each image, the mismatch between *all* pairs of images is minimized, in the least-squares sense. For each pair of images, the sky mismatch is computed *only* in the regions in which the two images overlap on the sky.

  This makes the "match" algorithm particularly useful for equalizing sky values in large mosaics in which one may have only pair-wise intersection of adjacent images without having a common intersection region (on the sky) in all images.

  Note that if the argument "match_down=True", matching will be done to the image with the lowest sky value, and if "match_down=False" it will be done to the image with the highest value (see *skymatch step arguments* for full details).
- The "global+match" algorithm combines the "global" and "match" methods. It uses the "global" algorithm to find a baseline sky value common to all input images and the "match" algorithm to equalize sky values among images. The direction of matching (to the lowest or highest) is again controlled by the "match_down" argument.

In the "local" and "global" methods, which find sky levels in each image, the calculation of the image statistics takes advantage of sigma clipping to remove contributions from isolated sources. This can work well for accurately determining the true sky level in images that contain semi-large regions of empty sky. The "match" algorithm, on the other hand, compares the *total* signal levels integrated over regions of overlap in each image pair. This method can produce

better results when there are no large empty regions of sky in the images. This method cannot measure the true sky level, but instead provides additive corrections that can be used to equalize the signal between overlapping images.

Examples

To get a better idea of the behavior of these different methods, the tables below show the results for two hypothetical sets of images. The first example is for a set of 6 images that form a 2x3 mosaic, with every image having overlap with its immediate neighbors. The first column of the table gives the actual (fake) sky signal that was imposed in each image, and the subsequent columns show the results computed by each method (i.e. the values of the resulting "level" attribute). All results are for the case where the step argument `match_down = True`, which means matching is done to the image with the lowest sky value. Note that these examples are for the highly simplistic case where each example image contains nothing but the constant sky value. Hence the sky computations are not affected at all by any source content and are therefore able to determine the sky values exactly in each image. Results for real images will of course not be so exact.

Sky	Local	Global	Match	Global+Match
100	100	100	0	100
120	120	100	20	120
105	105	100	5	105
110	110	100	10	110
105	105	100	5	105
115	115	100	15	115

- "local" finds the sky level of each image independently of the rest.
- "global" uses the minimum sky level found by "local" and applies it to all images.
- "match" with "match_down=True" finds the offset needed to match all images to the level of the image with the lowest sky level.
- "global+match" with "match_down=True" finds the offsets and global value needed to set all images to a sky level of zero. In this trivial example, the results are identical to the "local" method.

The second example is for a set of 7 images, where the first 4 form a 2x2 mosaic, with overlaps, and the second set of 3 images forms another mosaic, with internal overlap, but the 2 mosaics do *NOT* overlap one another.

Sky	Local	Global	Match	Global+Match
100	100	90	0	86.25
120	120	90	20	106.25
105	105	90	5	91.25
110	110	90	10	96.25
95	95	90	8.75	95
90	90	90	3.75	90
100	100	90	13.75	100

In this case, the "local" method again computes the sky in each image independently of the rest, and the "global" method sets the result for each image to the minimum value returned by "local". The matching results, however, require some explanation. With "match" only, all of the results give the proper offsets required to equalize the images contained within each mosaic, but the algorithm does not have the information needed to match the two (non-overlapping) mosaics to one another. Similarly, the "global+match" results again provide proper matching within each mosaic, but will leave an overall residual in one of the mosaics.

Limitations and Discussions

As alluded to above, the best sky computation method depends on the nature of the data in the input images. If the input images contain mostly compact, isolated sources, the "local" and "global" algorithms can do a good job at finding the true sky level in each image. If the images contain large, diffuse sources, the "match" algorithm is more appropriate, assuming of course there is sufficient overlap between images from which to compute the matching values. In the event there is not overlap between all of the images, as illustrated in the second example above, the "match" method can still provide useful results for matching the levels within each non-contigous region covered by the images, but will not provide a good overall sky level across all of the images. In these situations it is more appropriate to either process the non-contiguous groups independently of one another or use the "local" or "global" methods to compute the sky separately in each image. The latter option will of course only work well if the images are not domimated by extended, diffuse sources.

The primary reason for introducing the `skymatch` algorithm was to try to equalize the sky in large mosaics in which computation of the absolute sky is difficult, due to the presence of large diffuse sources in the image. As discussed above, the `skymatch` step accomplishes this by comparing the sky values in the overlap regions of each image pair. The quality of sky matching will obviously depend on how well these sky values can be estimated. True background may not be present at all in some images, in which case the computed "sky" may be the surface brightness of a large galaxy, nebula, etc.

Here is a brief list of possible limitations and factors that can affect the outcome of the matching (sky subtraction in general) algorithm:

- Because sky computation is performed on *flat-fielded* but *not distortion corrected* images, it is important to keep in mind that flat-fielding is performed to obtain correct surface brightnesses. Because the surface brightness of a pixel containing a point-like source will change inversely with a change to the pixel area, it is advisable to mask point-like sources through user-supplied mask files. Values different from zero in user-supplied masks indicate good data pixels. Alternatively, one can use the `upper` parameter to exclude the use of pixels containing bright objects when performing the sky computations.
- 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 (see *skymatch step arguments*).

Step Arguments

The `skymatch` step uses the following optional arguments:

General sky matching parameters:

`skymethod` (str, default='match')
: The sky computation algorithm to be used. Allowed values: `local`, `global`, `match`, `global+match`

`match_down` (boolean, default=True)
: Specifies whether the sky *differences* should be subtracted from images with higher sky values (`match_down` = `True`) 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`). **NOTE**: this argument only applies when `skymethod` is either `match` or `global+match`.

`subtract` (boolean, default=False)
Specifies whether the computed sky background values are to be subtracted from the images. The BKGSUB keyword (boolean) will be set in each output image to record whether or not the background was subtracted.

Image bounding polygon parameters:

`stepsize` (int, default=None)
Spacing between vertices of the images bounding polygon. The default value of `None` creates bounding polygons with four vertices corresponding to the corners of the image.

Sky statistics parameters:

`skystat` (str, default='mode')
Statistic to be used for sky background computations. Supported values are: `mean`, `mode`, `midpt`, and `median`.

`dqbits` (str, default='~DO_NOT_USE+NON_SCIENCE')
The DQ bit values from the input images' DQ arrays that should be considered "good" when building masks for sky computations. See DQ flag *Data Quality (DQ) Initialization* for details. The default value rejects pixels flagged as either 'DO_NOT_USE' or 'NON_SCIENCE' and considers all others to be good.

`lower` (float, default=None)
An optional value indicating the lower limit of usable pixel values for computing the sky. This value should be specified in the units of the input images.

`upper` (float, default=None)
An optional value indicating the upper limit of usable pixel values for computing the sky. This value should be specified in the units of the input images.

`nclip` (int, default=5)
The number of clipping iterations to use when computing sky values.

`lsig` (float, default=4.0)
Lower clipping limit, in sigma, used when computing the sky value.

`usig` (float, default=4.0)
Upper clipping limit, in sigma, used when computing the sky value.

`binwidth` (float, default=0.1)
Bin width, in sigma, used to sample the distribution of pixel values in order to compute the sky background using statistics that require binning, such as `mode` and `midpt`.

Reference File

The `skymatch` step does not use any reference files.

skymatch_step

The `skymatch_step` function (class name `SkyMatchStep`) is the top-level function used to call the skymatch operation from the Roman calibration pipeline. Roman step for sky matching.

class `romancal.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, 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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

```
class_alias = 'skymatch'
```

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:  ClassVar = []
```

```
spec = '\n # General sky matching parameters:\n skymethod = option(\'local\',
\'global\', \'match\', \'global+match\', default=\'match\') # sky computation
method\n match_down = boolean(default=True) # adjust sky to lowest measured value?\n
subtract = boolean(default=False) # subtract computed sky from image data?\n\n #
Image\'s bounding polygon parameters:\n stepsize = integer(default=None) # Max
vertex separation\n\n # Sky statistics parameters:\n skystat = option(\'median\',
\'midpt\', \'mean\', \'mode\', default=\'mode\') # sky statistics\n dqbits =
string(default=\'~DO_NOT_USE+NON_SCIENCE\') # "good" DQ bits\n lower =
float(default=None) # Lower limit of "good" pixel values\n upper =
float(default=None) # Upper limit of "good" pixel values\n nclip = integer(min=0,
default=5) # number of sky clipping iterations\n lsigma = float(min=0.0,
default=4.0) # Lower clipping limit, in sigma\n usigma = float(min=0.0, default=4.0)
# Upper clipping limit, in sigma\n binwidth = float(min=0.0, default=0.1) # Bin
width for \'mode\' and \'midpt\' `skystat`, in sigma\n '
```

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.

`romancal.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 *SkyImage* or *SkyGroup* objects.
- **skymethod** (*{'local', 'global+match', 'global', 'match'}, optional*) – Select the algorithm for sky computation:
  - **'local'** : compute sky background values of each input image or group of images (members of the same "exposure"). A single sky value is computed for each group of images.

Note: This setting is recommended when regions of overlap between images are dominated by "pure" sky (as opposed to extended, diffuse sources).

- **'global'** : compute a common sky value for all input images and groups of images. With this setting `local` will compute sky values for each input image/group, find the minimum sky value, and then it will set (and/or subtract) the sky value of each input image to this minimum value. This method *may* be useful when the input images have been already matched.
- **'match'** : compute differences in sky values between images and/or groups in (pair-wise) common sky regions. In this case the computed sky values will be relative (delta) to the sky computed in one of the input images whose sky value will be set to (reported to be) 0. This setting will "equalize" sky values between the images in large mosaics. However, this method is not recommended when used in conjunction with astrodrizzle because it computes relative sky values while `astrodrizzle` needs "absolute" sky values for median image generation and CR rejection.
- **'global+match'** : first use the **'match'** method to equalize sky values between images and then find a minimum "global" sky value amongst all input images.

Note: This is the *recommended* setting for images containing diffuse sources (e.g., galaxies, nebulae) covering significant parts of the image.

- **match_down** (*bool, optional*) – Specifies whether the sky *differences* should be subtracted from images with higher sky values (`match_down = 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`).

Note: This setting applies *only* when the `skymethod` parameter is either `'match'` or `'global+match'`.

- **subtract** (*bool (Default = False)*) – Subtract computed sky value from image data.

Raises

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.

Two new methods of sky subtraction have been introduced (compared to the standard `'local'`): `'global'` and `'match'`, as well as a combination of the two – `'global+match'`.

- The `'global'` method computes the minimum sky value across *all* input images and/or groups. That sky value is then considered to be the background in all input images.
- The `'match'` algorithm is somewhat similar to the traditional sky subtraction method (`skymethod = 'local'`) in the sense that it measures the sky independently in input images (or groups). The major differences are that, unlike the traditional method,

1. 'match' algorithm computes *relative* (delta) sky values with regard to the sky in a reference image chosen from the input list of images; *and*
2. Sky statistics are computed only in the part of the image that intersects other images.

This makes the 'match' sky computation algorithm particularly useful for "equalizing" sky values in large mosaics in which one may have only (at least) pair-wise intersection of images without having a common intersection region (on the sky) in all images.

The 'match' method works in the following way: for each pair of intersecting images, an equation is written that requires that average surface brightness in the overlapping part of the sky be equal in both images. The final system of equations is then solved for unknown background levels.

> **Warning:** The current algorithm is not capable of detecting cases where some subsets of intersecting images (from the input list of images) do not intersect at all with other subsets of intersecting images (except for the simple case when *single* images do not intersect any other images). In these cases the algorithm will find equalizing sky values for each intersecting subset of images and/or groups of images. However since these subsets of images do not intersect each other, sky will be matched only within each subset and the "inter-subset" sky mismatch could be significant.
>
> Users are responsible for detecting such cases and adjusting processing accordingly.

- The 'global+match' algorithm combines the 'match' and 'global' methods in order to overcome the limitation of the 'match' method described in the note above: it uses the 'global' algorithm to find a baseline sky value common to all input images and the 'match' algorithm to "equalize" sky values in the mosaic. Thus, the sky value of the "reference" image will be equal to the baseline sky value (instead of 0 in 'match' algorithm alone).

Remarks:

- *match()* works directly on *geometrically distorted* flat-fielded images thus avoiding the need to perform distortion correction on the input images.

  Initially, the footprint of a chip in an image is approximated by a 2D planar rectangle representing the borders of chip's distorted image. After applying distortion model to this rectangle and projecting it onto the celestial sphere, it is approximated by spherical polygons. Footprints of exposures and mosaics are computed as unions of such spherical polygons while overlaps of image pairs are found by intersecting these spherical polygons.

Limitations and Discussions:

Primary reason for introducing "sky match" algorithm was to try to equalize the sky in large mosaics in which computation of the "absolute" sky is difficult due to the presence of large diffuse sources in the image. As discussed above, *match()* accomplishes this by comparing "sky values" in a pair of images in the overlap region (that is common to both images). Quite obviously the quality of sky "matching" will depend on how well these "sky values" can be estimated. We use quotation marks around *sky values* because for some image "true" background may not be present at all and the measured sky may be the surface brightness of large galaxy, nebula, etc.

In the discussion below we will refer to parameter names in *SkyStats* and these parameter names may differ from the parameters of the actual skystat object passed to initializer of the *SkyImage*.

Here is a brief list of possible limitations/factors that can affect the outcome of the matching (sky subtraction in general) algorithm:

- Since sky subtraction is performed on *flat-fielded* but *not distortion corrected* images, it is important to keep in mind that flat-fielding is performed to obtain uniform surface brightness and not flux. This distinction is important for images that have not been distortion corrected. As a consequence, it is advisable that point-like sources be masked through the user-supplied mask files. Values different from

zero in user-supplied masks indicate "good" data pixels. Alternatively, one can use `upper` parameter to limit the use of bright objects in sky computations.

- Normally, distorted flat-fielded images contain cosmic rays. This algorithm does not perform CR cleaning. A possible way of minimizing the effect of the cosmic rays on sky computations is to use clipping (`nclip` > 0) and/or set `upper` parameter to a value larger than most of the sky background (or extended source) but lower than the values of most CR pixels.
- In general, clipping is a good way of eliminating "bad" pixels: pixels affected by CR, hot/dead pixels, etc. However, for images with complicated backgrounds (extended galaxies, nebulae, etc.), affected by CR and noise, clipping process may mask different pixels in different images. If variations in the background are too strong, clipping may converge to different sky values in different images even when factoring in the "true" difference in the sky background between the two images.
- In general images can have different "true" background values (we could measure it if images were not affected by large diffuse sources). However, arguments such as `lower` and `upper` will apply to all images regardless of the intrinsic differences in sky levels.

skyimage

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.

class romancal.skymatch.skyimage.**DataAccessor**

Base class for all data accessors. Provides a common interface to access data.

abstract get_data()

abstract get_data_shape()

abstract set_data(*data*)

Sets data.

Parameters
data (*numpy.ndarray*) – Data array to be set.

class romancal.skymatch.skyimage.**NDArrayInMemoryAccessor**(*data*)

Acessor for in-memory `numpy.ndarray` data.

get_data()

get_data_shape()

set_data(*data*)

Sets data.

Parameters
data (*numpy.ndarray*) – Data array to be set.

class romancal.skymatch.skyimage.**NDArrayMappedAccessor**(*data*, *tmpfile=None*, *prefix='tmp_skymatch_'*, *suffix='.npy'*, *tmpdir=''*)

Data accessor for arrays stored in temporary files.

get_data()

get_data_shape()

set_data(*data*)

Sets data.

Parameters

data (*numpy.ndarray*) – Data array to be set.

class romancal.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*, *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, sky statistics will be computed over the entire image.
- **delta** (*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.
- **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.

property id

Set or get *SkyImage*'s *id*.

While *id* can be of any type, it is preferable that *id* be of a type with nice string representation.

insert(*idx*, *value*)

Inserts a *SkyImage* into the group.

intersection(*skyimage*)

Compute intersection of this *SkyImage* object and another *SkyImage*, *SkyGroup*, or SphericalPolygon object.

Parameters

skyimage (SkyImage, SkyGroup, *SphericalPolygon*) – Another object that should be intersected with this *SkyImage*.

Returns

intersect_poly – A SphericalPolygon that is the intersection of this *SkyImage* and skyimage.

Return type
SphericalPolygon

property polygon

Get image's bounding polygon.

property radec

Get RA and DEC of the vertices of the bounding polygon as a `ndarray` of shape (N, 2) where N is the number of vertices + 1.

property sky

Sky background value. See *calc_sky* for more details.

class romancal.skymatch.skyimage.**SkyImage**(*image*, *wcs_fwd*, *wcs_inv*, *pix_area=1.0*, *convf=1.0*, *mask=None*, *id=None*, *skystat=None*, *stepsize=None*, *meta=None*, *reduce_memory_usage=True*)

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, NDArrayDataAccessor*) – A 2D array of image data or a `NDArrayDataAccessor`.
- **wcs_fwd** (*function*) – "forward" pixel-to-world transformation function.
- **wcs_inv** (*function*) – "inverse" world-to-pixel transformation function.
- **pix_area** (*float, optional*) – Average pixel's sky area.
- **convf** (*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, NDArrayDataAccessor*) – A 2D array or `NDArrayDataAccessor` of a 2D array that indicates which pixels in the input *image* should be used for sky computations (`1`) and which pixels should **not** be used for sky computations (`0`).
- **id** (*anything*) – The value of this parameter is simply stored within the *SkyImage* object. While it can be of any type, it is preferable that *id* be of a type with nice string representation.

- **skystat** (*callable, None, optional*) – A callable object that takes a either a 2D image (2D `numpy.ndarray`) or a list of pixel values (an 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, None, optional*) – Spacing between vertices of the image's bounding polygon. Default value of `None` creates bounding polygons with four vertices corresponding to the corners of the image.

- **meta** (*dict, None, optional*) – A dictionary of various items to be stored within the *SkyImage* object.

- **reduce_memory_usage** (*bool, optional*) – Indicates whether to attempt to minimize memory usage by attaching input `image` and/or `mask` `numpy.ndarray` arrays to file-mapped accessor. This has no effect when input parameters `image` and/or `mask` are already of `NDArrayDataAccessor` objects.

calc_bounding_polygon(*stepsize=None*)

Compute image's bounding polygon.

Parameters

stepsize (*int, None, optional*) – Indicates the maximum separation between two adjacent vertices of the bounding polygon along each side of the image. Corners of the image are included automatically. If `stepsize` is `None`, 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, 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`, sky statistics will be computed over the entire image.

- **delta** (*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`.

- **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.

property id

Set or get *SkyImage*'s *id*.

While *id* can be of any type, it is preferable that *id* be of a type with nice string representation.

property image

Set or get *SkyImage*'s image data array.

property image_shape

Get *SkyImage*'s image data shape.

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

property is_sky_valid

Indicates whether sky value was successfully computed. Must be set externally.

property mask

Set or get *SkyImage*'s mask data array or None.

property pix_area

Set or get mean pixel area.

property poly_area

Get bounding polygon area in srad units.

property polygon

Get image's bounding polygon.

property radec

Get RA and DEC of the vertices of the bounding polygon as a ndarray of shape (N, 2) where N is the number of vertices + 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.

property sky

Sky background value. See *calc_sky* for more details.

property skystat

Stores/retrieves a callable object that takes a either a 2D image (2D numpy.ndarray) or a list of pixel values (an 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.

skystatistics

The `skystatistics` module contains various statistical functions used by `skymatch`. The `skystatistics` module provides statistics computation class used by *match()* and *SkyImage*.

class romancal.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 while 'median' will compute the median of the distribution.
- **lower** (*float, None, optional*) – Lower limit of usable pixel values for computing the sky. This value should be specified in the units of the input image(s).
- **upper** (*float, 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, optional*) – A non-negative number of clipping iterations to use when computing the sky value.
- **lsig** (*float, optional*) – Lower clipping limit, in sigma, used when computing the sky value.
- **usig** (*float, optional*) – Upper clipping limit, in sigma, used when computing the sky value.
- **binwidth** (*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*) – A dictionary of optional arguments to be passed to `ImageStats`.

calc_sky(*data*)

Computes statistics on data.

Parameters

data (*numpy.ndarray*) – A numpy array of values for which the statistics needs to be computed.

Returns

statistics – A tuple of two values: (`skyvalue`, `npix`), where `skyvalue` is the statistics specified by the `skystat` parameter during the initialization of the *SkyStats* object and `npix` is the number of pixels used in computing the statistics reported in `skyvalue`.

Return type

tuple

region

The region module provides a polygon filling algorithm used by skymatch to create data masks. Polygon filling algorithm.

NOTE: Algorithm description can be found, e.g., here:
http://www.cs.rit.edu/~icss571/filling/how_to.html http://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/PolygonFilling.html

class romancal.skymatch.region.**Edge**(*name=None*, *start=None*, *stop=None*, *next=None*)

Edge representation

An edge has "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*)

property next

property start

property stop

property ymax

property ymin

class romancal.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**
- **(GET)** (*- Set the Global Edge Table*)
- **GET** (*- Set y to be the smallest y coordinate that has an entry in*)
- **empty** (*- Initialize the Active Edge Table (AET) to be*)
- **line** (*- For each scan*) –
  1. Add edges from GET to AET for which ymin==y
  2. Remove edges from AET fro which ymax==y
  3. Compute the intersection of the current scan line with all edges in the AET
  4. Sort on X of intersection point
  5. Set elements between pairs of X in the AET to the Edge's ID

update_AET(*y*, *AET*)

Update the Active Edge Table (AET)

Add edges from GET to AET for which ymin of the edge is equal to the y of the scan line. Remove edges from AET for which ymax of the edge is equal to y of the scan line.

class romancal.skymatch.region.**Region**(*rid*, *coordinate_system*)

Base class for regions.

Parameters

- **rid** (*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

romancal.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 romancal.skymatch.**SkyMatchStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

SkyMatchStep: Subtraction or equalization of sky background in science images.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.

Attributes Documentation

class_alias = 'skymatch'

reference_file_types: ClassVar = []

spec

```
# General sky matching parameters:
skymethod = option('local', 'global', 'match', 'global+match', default='match')
↪# sky computation method
match_down = boolean(default=True) # adjust sky to lowest measured value?
subtract = boolean(default=False) # subtract computed sky from image data?

# Image's bounding polygon parameters:
stepsize = integer(default=None) # Max vertex separation

# Sky statistics parameters:
skystat = option('median', 'midpt', 'mean', 'mode', default='mode') # sky␣
↪statistics
dqbits = string(default='~DO_NOT_USE+NON_SCIENCE') # "good" DQ bits
lower = float(default=None) # Lower limit of "good" pixel values
upper = float(default=None) # Upper limit of "good" pixel values
nclip = integer(min=0, default=5) # number of sky clipping iterations
lsigma = float(min=0.0, default=4.0) # Lower clipping limit, in sigma
usigma = float(min=0.0, default=4.0) # Upper clipping limit, in sigma
binwidth = float(min=0.0, default=0.1) # Bin width for 'mode' and 'midpt'␣
↪`skystat`, in sigma
```

Methods Documentation

process(*input*)

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

Class Inheritance Diagram

1.4.15 Outlier Detection

Description

Classes
romancal.outlier_detection.OutlierDetectionStep

Aliases
outlier_detection, outlier_detection_scaled, outlier_detection_stack

Processing multiple datasets together allows for the identification of bad pixels or cosmic-rays that remain in each of the input images, many times at levels which were not detectable by the *jump* step. The `outlier_detection` step implements the following algorithm to identify and flag any remaining cosmic-rays or other artifacts left over from previous calibrations:

- build a stack of input data
  - all inputs will need to have the same WCS, which is done by *romancal.tweakreg.TweakRegStep*), since outlier detection assumes the same flux for each point on the sky, and variations from one image to the next would indicate a spurious signal
  - 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.

Step Arguments

The `outlier_detection` step has the following optional arguments that control the behavior of the processing:

`--weight_type` (string, default='exptime')
The type of data weighting to use during resampling the images for creating the median image used for detecting outliers; options are `'ivm'`, `'exptime'`, and `None` (see *Weighting types* for details).

`--pixfrac` (float, default=1.0)
Fraction by which input pixels are "shrunk" before being drizzled onto the output image grid, given as a real number between 0 and 1. This specifies the size of the footprint, or "dropsize", of a pixel in units of the input pixel size. If `pixfrac` is set to less than 0.001, the `kernel` parameter will be reset to `'point'` for more efficient processing. In the step of drizzling each input image onto a separate output image, the default value of 1.0 is best in order to ensure that each output drizzled image is fully populated with pixels from the input image. Valid values range from 0.0 to 1.0.

`--kernel` (string, default='square')
This parameter specifies the form of the kernel function used to distribute flux onto the separate output images, for the initial separate drizzling operation only. The value options for this parameter include:

- `'square'`: original classic drizzling kernel

- `'tophat'`: this kernel is a circular "top hat" shape of width `pixfrac`. It effects only output pixels within a radius of `pixfrac/2` from the output position.
- `'lanczos3'`: a Lanczos style kernel, extending a radius of 3 pixels from the center of the detection. The Lanczos kernel is a damped and bounded form of the "sinc" interpolator, and is very effective for resampling single images when `scale=pixfrac=1`. It leads to less resolution loss than other kernels, and typically results in reduced correlated noise in outputs.

Warning: The `'lanczos3'` kernel tends to result in much slower processing as compared to other kernel options. This option should never be used for `pixfrac != 1.0`, and is not recommended for `scale!=1.0`.

`--fillval` (string, default='INDEF')
The value for this parameter is to be assigned to the output pixels that have zero weight or which do not receive flux from any input pixels during drizzling. This parameter corresponds to the `fillval` parameter of the `drizzle` task. If the default of `None` is used, and if the weight in both the input and output images for a given pixel are zero, then the output pixel will be set to the value it would have had if the input had a non-zero weight. Otherwise, if a numerical value is provided (e.g. 0), then these pixels will be set to that numerical value. Any floating-point value, given as a string, is valid. A value of 'INDEF' will use the last zero weight flux.

`--nlow` (integer, default=0)
The number of low values in each pixel stack to ignore when computing the median value.

`--nhigh` (integer, default=0)
The number of high values in each pixel stack to ignore when computing the median value.

`--maskpt` (float, default=0.7)
Percentage of weight image values below which they are flagged as bad and rejected from the median image. Valid values range from 0.0 to 1.0.

`--grow` (integer, default=1)
The distance, in pixels, beyond the limit set by the rejection algorithm being used, for additional pixels to be rejected in an image.

`--snr` (string, default='4.0 3.0')
The signal-to-noise values to use for bad pixel identification. Since cosmic rays often extend across several pixels the user must specify two cut-off values for determining whether a pixel should be masked: the first for detecting the primary cosmic ray, and the second (typically lower threshold) for masking lower-level bad pixels adjacent to those found in the first pass. Valid values are a pair of floating-point values in a single string.

`--scale` (string, default='0.5 0.4')
The scaling factor applied to derivative used to identify bad pixels. Since cosmic rays often extend across several pixels the user must specify two cut-off values for determining whether a pixel should be masked: the first for detecting the primary cosmic ray, and the second (typically lower threshold) for masking lower-level bad pixels adjacent to those found in the first pass. Valid values are a pair of floating-point values in a single string.

`--backg` (float, default=0.0)
User-specified background value (scalar) to subtract during final identification step of outliers in `driz_cr` computation.

`--kernel_size` (string, default='7 7')
Size of kernel to be used during resampling of the data (i.e. when `resample_data=True`).

`--save_intermediate_results` (boolean, default=False)
Specifies whether or not to write out intermediate products such as median image or resampled individual input exposures to disk. Typically, only used to track down problems with final results when too many or too few pixels are flagged as outliers.

--resample_data (boolean, default=True)
Specifies whether or not to resample the input images when performing outlier detection.

--good_bits (string, default=0)
The DQ bit values from the input image DQ arrays that should be considered 'good' when creating masks of bad pixels during outlier detection when resampling the data. See Roman's Data Quality Flags for details.

--allowed_memory (float, default=None)
Specifies the fractional amount of free memory to allow when creating the resampled image. If `None`, the environment variable `DMODEL_ALLOWED_MEMORY` is used. If not defined, no check is made. If the resampled image would be larger than specified, an `OutputTooLargeError` exception will be generated. For example, if set to `0.5`, only resampled images that use less than half the available memory can be created.

--in_memory (boolean, default=False)
Specifies whether or not to keep all intermediate products and datamodels in memory at the same time during the processing of this step. If set to `False`, all input and output data will be written to disk at the start of the step (as much as `roman_datamodels` will allow, anyway), then read in to memory only when accessed. This results in a much lower memory profile at the expense of file I/O, which can allow large mosaics to process in more limited amounts of memory.

Weighting types

`weight_type` specifies the type of weighting image to apply with the bad pixel mask for the final drizzle step. The options for this parameter include:

- `ivm`: allows the user to either supply their own inverse-variance weighting map, or allow `drizzle` to generate one automatically on-the-fly during the final drizzle step. This parameter option may be necessary for specific purposes. For example, to create a drizzled weight file for software such as `SExtractor`, it is expected that a weight image containing all of the background noise sources (sky level, read-noise, dark current, etc), but not the Poisson noise from the objects themselves will be available. The user can create the inverse variance images and then specify their names using the `input` parameter for `drizzle` to specify an '@file'. This would be a single `ASCII` file containing the list of input calibrated exposure filenames (one per line), with a second column containing the name of the `IVM` file corresponding to each calibrated exposure. Each `IVM` file must have the same file format as the input file.
- `exptime`: the images will be weighted according to their exposure time, which is the standard behavior for drizzle. This weighting is a good approximation in the regime where the noise is dominated by photon counts from the sources, while contributions from sky background, read-noise and dark current are negligible. This option is provided as the default since it produces reliable weighting for all types of data.
- `None`: In this case, a bit mask will be generated based on the DQ array and a bit flag set to 0 (i.e. `GOOD`; see Roman's Data Quality Flags for details).

OutlierDetectionStep

This module provides the sole interface to all methods of performing outlier detection on Roman observations. The outlier detection algorithm used for WFI data is implemented in *`OutlierDetection`* and described in *Outlier Detection Algorithm*.

Note: Whether the data are being provided in an association file or as a list of ASDF filenames, they must always be wrapped with a *`ModelContainer`*, which will handle and read in the input properly.

On successful completion, this step will return the input models with DQ arrays updated with flags for identified outliers.

romancal.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 romancal.outlier_detection.outlier_detection_step.**OutlierDetectionStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Flag outlier bad pixels and cosmic rays in DQ array of each input image.

Input images can be listed in an input association file or already wrapped with a ModelContainer. DQ arrays are modified in place.

Parameters
input_data (*ModelContainer*) – A *ModelContainer* object.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(input_models)	Perform outlier detection processing on input data.

Attributes Documentation

class_alias = 'outlier_detection'

spec

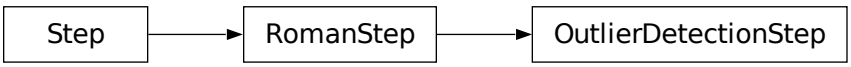
```
weight_type = option('ivm','exptime',default='ivm') # Weighting type to use to␣
↪create the median image
pixfrac = float(default=1.0) # Fraction by which input pixels are shrunk before␣
↪being drizzled onto the output image grid
kernel = string(default='square') # Shape of the kernel used for flux␣
↪distribution onto output images
fillval = string(default='INDEF') # Value assigned to output pixels that have␣
↪zero weight or no flux during drizzling
nlow = integer(default=0) # The number of low values in each pixel stack to␣
↪ignore when computing the median value
nhigh = integer(default=0) # The number of high values in each pixel stack to␣
↪ignore when computing the median value
maskpt = float(default=0.7) # Percentage of weight image values below which␣
↪they are flagged as bad pixels
grow = integer(default=1) # The distance beyond the rejection limit for␣
↪additional pixels to be rejected in an image
snr = string(default='5.0 4.0') # The signal-to-noise values to use for bad␣
↪pixel identification
scale = string(default='1.2 0.7') # The scaling factor applied to derivative␣
↪used to identify bad pixels
backg = float(default=0.0) # User-specified background value to subtract during␣
↪final identification step
kernel_size = string(default='7 7') # Size of kernel to be used during␣
↪resampling of the data
save_intermediate_results = boolean(default=False) # Specifies whether or not␣
↪to write out intermediate products to disk
resample_data = boolean(default=True) # Specifies whether or not to resample␣
↪the input images when performing outlier detection
good_bits = string(default="~DO_NOT_USE+NON_SCIENCE")  # DQ bit value to be␣
↪considered 'good'
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image
in_memory = boolean(default=False) # Specifies whether or not to keep all␣
↪intermediate products and datamodels in memory
```

Methods Documentation

process(*input_models*)

Perform outlier detection processing on input data.

Class Inheritance Diagram



Outlier Detection Algorithm

This module serves as the interface for applying `outlier_detection` to direct image observations. The code implements the basic outlier detection algorithm used with JWST data, but adapted to Roman.

Specifically, this routine performs the following operations:

1. Extract parameter settings from input model and merge them with any user-provided values.
   - See *outlier detection arguments* for the full list of parameters.
2. By default, resample all input images.
   - The resampling step starts by computing an output WCS that is large enough to encompass all the input images.
   - All 18 detectors from the *same exposure* will get resampled onto this output WCS to create a mosaic of all the chips for that exposure. This product is referred to as a "grouped mosaic" since it groups all the chips from the same exposure into a single image.
   - Each dither position will result in a separate grouped mosaic, so only a single exposure ever contributes to each pixel in these mosaics.
   - The `fillval` parameter specifies what value to use in the output resampled image for any pixel which has no valid contribution from any input exposure. The default value of `INDEF` indicates that the value from the last exposure will be used, while a value of 0 would result in holes.
   - The resampling can be controlled with the `pixfrac`, `kernel` and `weight_type` parameters.
   - The `pixfrac` indicates the fraction by which input pixels are "shrunk" before being drizzled onto the output image grid, given as a real number between 0 and 1. This specifies the size of the footprint, or "dropsize", of a pixel in units of the input pixel size.
   - The `kernel` specifies the form of the kernel function used to distribute flux onto the separate output images.
   - The `weight_type` indicates the type of weighting image to apply with the bad pixel mask. Available options are `ivm` (default) for computing and using an inverse-variance map and `exptime` for weighting by the exposure time.
   - The `good_bits` parameter specifies what DQ values from the input exposure should be used when resampling to create the output mosaic. Any pixel with a DQ value not included in this value (or list of values) will be ignored when resampling.

- Resampled images will be written out to disk with suffix `_outlier_i2d` by default.
- **If resampling is turned off** through the use of the `resample_data` parameter, a copy of the unrectified input images (as a ModelContainer) will be used for subsequent processing.

3. Create a median image from all grouped observation mosaics.
   - The median image is created by combining all grouped mosaic images or non-resampled input data pixel-by-pixel.
   - The `nlow` and `nhigh` parameters specify how many low and high values to ignore when computing the median for any given pixel.
   - The `maskpt` parameter sets the percentage of the weight image values to use, and any pixel with a weight below this value gets flagged as "bad" and ignored when resampled.
   - The `grow` parameter sets the width, in pixels, beyond the limit set by the rejection algorithm being used, for additional pixels to be rejected in an image.
   - The median image is written out to disk as `_<asn_id>_median` by default.
4. By default, the median image is blotted back (inverse of resampling) to match each original input image.
   - Blotted images are written out to disk as `_<asn_id>_blot` by default.
   - **If resampling is turned off**, the median image is compared directly to each input image.
5. Perform statistical comparison between blotted image and original image to identify outliers.
   - This comparison uses the original input images, the blotted median image, and the derivative of the blotted image to create a cosmic ray mask for each input image.
   - The derivative of the blotted image gets created using the blotted median image to compute the absolute value of the difference between each pixel and its four surrounding neighbors with the largest value being the recorded derivative.
   - These derivative images are used to flag cosmic rays and other blemishes, such as moving object trails. Where the difference is larger than can be explained by noise statistics, the flattening effect of taking the median, or an error in the shift (the latter two effects are estimated using the image derivative), the suspect pixel is masked.
   - The `backg` parameter specifies a user-provided value to be used as the background estimate. This gets added to the background-subtracted blotted image to attempt to match the original background levels of the original input mosaic so that cosmic-rays (bad pixels) from the input mosaic can be identified more easily as outliers compared to the blotted mosaic.
   - Cosmic rays are flagged using the following rule:

$$|image\_input - image\_blotted| > scale * image\_deriv + SNR * noise$$

   - The `scale` is defined as the multiplicative factor applied to the derivative which is used to determine if the difference between the data image and the blotted image is large enough to require masking.
   - The `noise` is calculated using a combination of the detector read noise and the poisson noise of the blotted median image plus the sky background.
   - The user must specify two cut-off signal-to-noise values using the `snr` parameter for determining whether a pixel should be masked: the first for detecting the primary cosmic ray, and the second for masking lower-level bad pixels adjacent to those found in the first pass. Since cosmic rays often extend across several pixels, the adjacent pixels make use of a slightly lower SNR threshold.
6. Update input data model DQ arrays with mask of detected outliers.

Memory Model for Outlier Detection Algorithm

The outlier detection algorithm can end up using massive amounts of memory depending on the number of inputs, the size of each input, and the size of the final output product. Specifically,

1. The input `ModelContainer` all input exposures would have been kept open in memory to make processing more efficient.
2. The initial resample step creates an output product for EACH input that is the same size as the final output product, which for imaging modes can span all chips in the detector while also accounting for all dithers. For some Level 3 products, each resampled image can be on the order of 2Gb or more.
3. The median combination step then needs to have all pixels at the same position on the sky in memory in order to perform the median computation. The simplest implementation for this step requires keeping all resampled outputs fully in memory at the same time.

Many Level 3 products only include a modest number of input exposures that can be processed using less than 32Gb of memory at a time. However, there are a number of ways this memory limit can be exceeded. This has been addressed by implementing an overall memory model for the outlier detection that includes options to minimize the memory usage at the expense of file I/O. The control over this memory model happens with the use of the `in_memory` parameter. The full impact of this parameter during processing includes:

1. The `save_open` parameter gets set to `False` when opening the input *ModelContainer* object. This forces all input models in the input *ModelContainer* to get written out to disk. It then uses the filename of the input model during subsequent processing.
2. The `in_memory` parameter gets passed to the *ResampleStep* to set whether or not to keep the resampled images in memory or not. By default, the outlier detection processing sets this parameter to `False` so that each resampled image gets written out to disk.
3. Computing the median image works section-by-section by only keeping 1Mb of each input in memory at a time. As a result, only the final output product array for the final median image along with a stack of 1Mb image sections are kept in memory.
4. The final resampling step also avoids keeping all inputs in memory by only reading each input into memory 1 at a time as it gets resampled onto the final output product.

These changes result in a minimum amount of memory usage during processing at the obvious expense of reading and writing the products from disk.

romancal.outlier_detection.outlier_detection Module

Primary code for performing outlier detection on Roman observations.

Functions

flag_cr(sci_image, blot_image[, snr, scale, ...])	Masks outliers in science image by updating DQ in-place
abs_deriv(array)	Take the absolute derivative of a numpy array.

flag_cr

romancal.outlier_detection.outlier_detection.**flag_cr**(*sci_image*, *blot_image*, *snr='5.0 4.0'*, *scale='1.2 0.7'*, *backg=0*, *resample_data=True*, ***kwargs*)

Masks outliers in science image by updating DQ in-place

Mask blemishes in dithered data by comparing a science image with a model image and the derivative of the model image.

Parameters

- **sci_image** (*ImageModel*) – the science data
- **blot_image** (*ImageModel*) – the blotted median image of the dithered science frames
- **snr** (*str*) – Signal-to-noise ratio
- **scale** (*str*) – scaling factor applied to the derivative
- **backg** (*float*) – Background value (scalar) to subtract
- **resample_data** (*bool*) – Boolean to indicate whether blot_image is created from resampled, dithered data or not

abs_deriv

romancal.outlier_detection.outlier_detection.**abs_deriv**(*array*)

Take the absolute derivative of a numpy array.

Classes

OutlierDetection(input_models, **pars)	Main class for performing outlier detection.

OutlierDetection

class romancal.outlier_detection.outlier_detection.**OutlierDetection**(*input_models*, ***pars*)

Bases: 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** (*ModelContainer*) – A *ModelContainer* object containing the data to be processed.
- **pars** (*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.
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.

create_median(*resampled_models*)

Create a median image from the singly resampled images.

Notes

This version is simplified from astrodrizzle's version in the following ways: - type of combination: fixed to 'median' - 'minmed' not implemented as an option

detect_outliers(*blot_models*)

Flag DQ array for cosmic rays in input images.

The science frame in each ImageModel in self.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

blot_models (*JWST ModelContainer object*) – data model container holding ImageModels of the median output frame blotted back to the wcs and frame of the ImageModels in input_models

Returns

The dq array in each input model is modified in place

Return type

None

do_detection()

Flag outlier pixels in DQ of input images.

Class Inheritance Diagram

Examples

Whether the data are contained in a list of ASDF files or provided as an ASN file, the `ModelContainer` class must be used to properly handle the data that will be used in the outlier detection step.

1. To run the outlier detection step (with the default parameters) on a list of 2 ASDF files named `"img_1.asdf"` and `"img_2.asdf"`:

```
from romancal.outlier_detection import OutlierDetectionStep
from romancal.datamodels import ModelContainer
# read the file list into a ModelContainer object
mc = ModelContainer(["img_1.asdf", "img_2.asdf"])
step = OutlierDetectionStep()
step.process(mc)
```

2. To run the outlier detection step (with the default parameters) on an ASN file called "asn_sample.json" with the following content:

```
{
        "asn_type": "None",
        "asn_rule": "DMS_ELPP_Base",
        "version_id": null,
        "code_version": "0.9.1.dev28+ge987cc9.d20230106",
        "degraded_status": "No known degraded exposures in association.",
        "program": "noprogram",
        "constraints": "No constraints",
        "asn_id": "a3001",
        "target": "none",
        "asn_pool": "test_pool_name",
        "products": [
                {
                "name": "files.asdf",
                "members": [
                        {
                        "expname": "img_1.asdf",
                        "exptype": "science"
                        },
                        {
                        "expname": "img_2.asdf",
                        "exptype": "science"
                        }
                ]
                }
        ]
}
```

```
from romancal.outlier_detection import OutlierDetectionStep
from romancal.datamodels import ModelContainer
# read the file list into a ModelContainer object
mc = ModelContainer("asn_sample.json")
step = OutlierDetectionStep()
step.process(mc)
```

3. To run the outlier detection step (with the default parameters) on an ASN file called "asn_sample.json" (the files listed in the association file must have been processed through `TweakRegStep`) using the command line:

```
strun --disable-crds-steppar  --suffix='cal'  romancal.step.
↪OutlierDetectionStep asn_sample.json
```

romancal.outlier_detection Package

Classes

OutlierDetectionStep([name, parent, ...])	Flag outlier bad pixels and cosmic rays in DQ array of each input image.

OutlierDetectionStep

class romancal.outlier_detection.**OutlierDetectionStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Flag outlier bad pixels and cosmic rays in DQ array of each input image.

Input images can be listed in an input association file or already wrapped with a ModelContainer. DQ arrays are modified in place.

Parameters
input_data (*ModelContainer*) – A *ModelContainer* object.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
spec

Methods Summary

process(input_models)	Perform outlier detection processing on input data.

Attributes Documentation

class_alias = 'outlier_detection'

spec

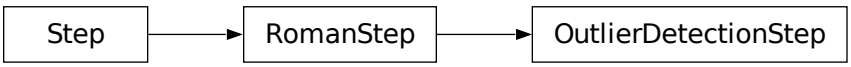
```
weight_type = option('ivm','exptime',default='ivm') # Weighting type to use to␣
↪create the median image
pixfrac = float(default=1.0) # Fraction by which input pixels are shrunk before␣
↪being drizzled onto the output image grid
kernel = string(default='square') # Shape of the kernel used for flux␣
↪distribution onto output images
fillval = string(default='INDEF') # Value assigned to output pixels that have␣
↪zero weight or no flux during drizzling
nlow = integer(default=0) # The number of low values in each pixel stack to␣
↪ignore when computing the median value
nhigh = integer(default=0) # The number of high values in each pixel stack to␣
↪ignore when computing the median value
maskpt = float(default=0.7) # Percentage of weight image values below which␣
↪they are flagged as bad pixels
grow = integer(default=1) # The distance beyond the rejection limit for␣
↪additional pixels to be rejected in an image
snr = string(default='5.0 4.0') # The signal-to-noise values to use for bad␣
↪pixel identification
scale = string(default='1.2 0.7') # The scaling factor applied to derivative␣
↪used to identify bad pixels
backg = float(default=0.0) # User-specified background value to subtract during␣
↪final identification step
kernel_size = string(default='7 7') # Size of kernel to be used during␣
↪resampling of the data
save_intermediate_results = boolean(default=False) # Specifies whether or not␣
↪to write out intermediate products to disk
resample_data = boolean(default=True) # Specifies whether or not to resample␣
↪the input images when performing outlier detection
good_bits = string(default="~DO_NOT_USE+NON_SCIENCE")  # DQ bit value to be␣
↪considered 'good'
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image
in_memory = boolean(default=False) # Specifies whether or not to keep all␣
↪intermediate products and datamodels in memory
```

Methods Documentation

process(*input_models*)

Perform outlier detection processing on input data.

Class Inheritance Diagram



1.4.16 Resample

Description

Classes

romancal.resample.ResampleStep

Alias

resample

This routine will resample each input 2D image based on the WCS and distortion information, and will combine multiple resampled images into a single undistorted product. The distortion information should have been incorporated into the image using the *assign_wcs* step.

The `resample` step can take:

- a single 2D input image (in the format of either a string with the full path and filename of an ASDF file or a Roman Datamodel/*ModelContainer*);
- an association table (in JSON format).

The parameters for the drizzle operation itself are set by *set_drizzle_defaults()*. 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 during the *ResampleStep* instantiation, only basic information is set.

The output product is determined by using the WCS information of all inputs, even if it is just a single image. The output WCS defines a field-of-view that encompasses the undistorted footprints on the sky of all the input images with the same orientation and plate scale as the first listed input image.

This step uses the interface to the C-based `cdriz` routine to do the resampling via the drizzle method (Fruchter and Hook, PASP 2002). The input-to-output pixel mapping is determined via a mapping function derived from the WCS of each input image and the WCS of the defined output product. The mapping function is created by *reproject()* and passed on to `cdriz` to drive the actual drizzling to create the output product.

Context Image

In addition to the resampled image data, resample step also creates a "context image" stored in the `con` attribute in the output data model. Each pixel in the context image is a bit field that encodes information about which input image has contributed to the corresponding pixel in the resampled data array. Context image uses 32 bit integers to encode this information and hence it can keep track of 32 input images at most.

For any given pixel, the first bit corresponds to the first input image, the second bit corrsponds to the second input image, and so on. If the number of input images is larger than 32, then it is necessary to have multiple context images ("planes") to hold information about all input images with the first plane encoding which of the first 32 images (indexed from 0 through 32) contributed to the output data pixel, second plane representing next 32 input images (indexed from 33 through 64), etc. For this reason, the context image is a 3D array of type `numpy.int32` and shape `(np, ny, nx)`, where `nx` and `ny` are the dimensions of the image's data and `np` is the number of "planes", which is equal to `(number of input images - 1) // 32 + 1`. If a bit at position `k` in a pixel with coordinates `(p, y, x)` is 0 then input image number `32 * p + k` (0-indexed) did not contribute to the output data pixel with array coordinates `(y, x)`, and if that bit is 1 then input image number `32 * p + k` did contribute to the pixel `(y, x)` in the resampled image.

As an example, let's assume we have 8 input images. Then, when `con` pixel values are displayed using binary representation (and decimal in parenthesis), one could see values like this:

```
00000001 (1) - only first input image contributed to this output pixel;
00000010 (2) - 2nd input image contributed;
00000100 (4) - 3rd input image contributed;
10000000 (128) - 8th input image contributed;
10000100 (132=128+4) - 3rd and 8th input images contributed;
11001101 (205=1+4+8+64+128) - input images 1, 3, 4, 7, 8 have contributed
to this output pixel.
```

In order to test if a specific input image contributed to an output pixel, one needs to use bitwise operations. Using the example above, to test whether input images number 4 and 5 have contributed to the output pixel whose corresponding `con` value is 205 (11001101 in binary form) we can do the following:

```
>>> bool(205 & (1 << (5 - 1)))  # (205 & 16) = 0 (== 0 => False): did NOT contribute
False
>>> bool(205 & (1 << (4 - 1)))  # (205 & 8) = 8 (!= 0 => True): did contribute
True
```

In general, to get a list of all input images that have contributed to an output resampled pixel with image coordinates `(x, y)`, and given a context array `con`, one can do something like this:

```
>>> import numpy as np
>>> np.flatnonzero([v & (1 << k) for v in con[:, y, x] for k in range(32)])
```

For convenience, this functionality was implemented in the *decode_context()* function.

References

- Fruchter and Hook, PASP 2002: full description of the drizzling algorithm.
- Casertano et al., AJ 2000 (Appendix A2): description of the inverse variance map method.
- DrizzlePac Handbook: description of the drizzle parameters and other useful drizzle-related resources.

Step Arguments

The `resample` step has the following optional arguments that control the behavior of the processing and the characteristics of the resampled image.

`--pixfrac` (float, default=1.0)
The fraction by which input pixels are "shrunk" before being drizzled onto the output image grid, given as a real number between 0 and 1.

`--kernel` (str, default='square')
The form of the kernel function used to distribute flux onto the output image. Available kernels are `square`, `gaussian`, `point`, `tophat`, `turbo`, `lanczos2`, and `lanczos3`.

`--pixel_scale_ratio` (float, default=1.0)
Ratio of input to output pixel scale. A value of 0.5 means the output image would have 4 pixels sampling each input pixel. Ignored when `pixel_scale` or `output_wcs` are provided.

`--pixel_scale` (float, default=None)
Absolute pixel scale in `arcsec`. When provided, overrides `pixel_scale_ratio`. Ignored when `output_wcs` is provided.

`--rotation` (float, default=None)
Position angle of output image's Y-axis relative to North. A value of 0.0 would orient the final output image to be North up. The default of `None` specifies that the images will not be rotated, but will instead be resampled in the default orientation for the camera with the x and y axes of the resampled image corresponding approximately to the detector axes. Ignored when `pixel_scale` or `output_wcs` are provided.

`--crpix` (tuple of float, default=None)
Position of the reference pixel in the image array in the `x, y` order. If `crpix` is not specified, it will be set to the center of the bounding box of the returned WCS object. When supplied from command line, it should be a comma-separated list of floats. Ignored when `output_wcs` is provided.

`--crval` (tuple of float, default=None)
Right ascension and declination of the reference pixel. Automatically computed if not provided. When supplied from command line, it should be a comma-separated list of floats. Ignored when `output_wcs` is provided.

`--output_shape` (tuple of int, default=None)
Shape of the image (data array) using "standard" `nx` first and `ny` second (as opposite to the `numpy.ndarray` convention - `ny` first and `nx` second). This value will be assigned to `pixel_shape` and `array_shape` properties of the returned WCS object. When supplied from command line, it should be a comma-separated list of integers `nx, ny`.

Note: Specifying `output_shape` *is required* when the WCS in `output_wcs` does not have `bounding_box` property set.

`--output_wcs` (str, default='')
File name of a `ASDF` file with a GWCS stored under the `"wcs"` key under the root of the file. The output image size is determined from the bounding box of the WCS (if any). Argument `output_shape` overrides computed image size and it is required when output WCS does not have `bounding_box` property set.

Note: When `output_wcs` is specified, WCS-related arguments such as `pixel_scale_ratio`, `pixel_scale`, `rotation`, `crpix`, and `crval` will be ignored.

`--fillval` (str, default='INDEF')
The value to assign to output pixels that have zero weight or do not receive any flux from any input pixels during drizzling.

`--weight_type` (str, default='ivm')
The weighting type for each input image. If `weight_type=ivm` (the default), the scaling value will be determined per-pixel using the inverse of the read noise (VAR_RNOISE) array stored in each input image. If the VAR_RNOISE array does not exist, the variance is set to 1 for all pixels (equal weighting). If `weight_type=exptime`, the scaling value will be set equal to the exposure time found in the image header.

`--single` (bool, default=False)
If set to `True`, resample each input image into a separate output. If `False` (the default), each input is resampled additively (with weights) to a common output

`--blendheaders` (bool, default=True)
Blend metadata from all input images into the resampled output image.

`--allowed_memory` (float, default=None)
Specifies the fractional amount of free memory to allow when creating the resampled image. If `None`, the environment variable `DMODEL_ALLOWED_MEMORY` is used. If not defined, no check is made. If the resampled image would be larger than specified, an `OutputTooLargeError` exception will be generated.

For example, if set to `0.5`, only resampled images that use less than half the available memory can be created.

`--in_memory` (bool, default=True)
If set to `False`, write output datamodel to disk.

`--good_bits` (str, default='~DO_NOT_USE+NON_SCIENCE')
Specifies the bits to use when creating the resampling mask. Either a single bit value or a combination of them can be provided. If the string starts with a tilde (), then the provided bit(s) will be excluded when creating the resampling mask. The default value is to exclude pixels flagged with `DO_NOT_USE` and `NON_SCIENCE`.

The bit value can be provided in a few different ways, but always as a string type. For example, if the user deems OK to use pixels with low QE and highly nonlinear, then any of the ways listed below will work to set `good_bits`:

- `good_bits = 'LOW_QE+NON_LINEAR'` (concatenated DQ flag labels);
- `good_bits = '8192+65536'` (concatenated DQ flag bit values);
- `good_bits = '8192,65536'` (comma-separated DQ flag bit values);
- `good_bits = '73728'` (sum of DQ flag bit values).

Note: Adding a tilde () to the beginning of the string will flip the bit(s) and actually `exclude` the provided bit(s). This is the same as providing the bad bits instead of the good bits.

Python Step Interface: ResampleStep()

romancal.resample.resample_step Module

Classes

ResampleStep([name, parent, config_file, ...])	Resample input data onto a regular grid using the drizzle algorithm.

ResampleStep

class romancal.resample.resample_step.**ResampleStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Resample input data onto a regular grid using the drizzle algorithm.

Note: When supplied via `output_wcs`, a custom WCS overrides other custom WCS parameters such as `output_shape` (now computed from by `output_wcs.bounding_box`), `crpix`

Parameters

input (str, `roman_datamodels.datamodels.DataModel`, or *ModelContainer*) – If a string is provided, it should correspond to either a single ASDF filename or an association filename. Alternatively, a single DataModel instance can be provided instead of an ASDF filename. Multiple files can be processed via either an association file or wrapped by a *ModelContainer*.

Returns

A mosaic datamodel with the final output frame.

Return type

`roman_datamodels.datamodels.MosaicModel`

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.
set_drizzle_defaults()	Set the default parameters for drizzle.
update_phot_keywords(model)	Update pixel scale keywords

Attributes Documentation

class_alias = 'resample'

reference_file_types: ClassVar = []

spec

```
pixfrac = float(default=1.0) # change back to None when drizpar reference files␣
↪are updated
kernel = string(default='square') # change back to None when drizpar reference␣
↪files are updated
fillval = string(default='INDEF' ) # change back to None when drizpar reference␣
↪files are updated
weight_type = option('ivm', 'exptime', None, default='ivm')  # change back to␣
↪None when drizpar ref update
output_shape = int_list(min=2, max=2, default=None)  # [x, y] order
crpix = float_list(min=2, max=2, default=None)
crval = float_list(min=2, max=2, default=None)
rotation = float(default=None)
pixel_scale_ratio = float(default=1.0) # Ratio of input to output pixel scale
pixel_scale = float(default=None) # Absolute pixel scale in arcsec
output_wcs = string(default='')  # Custom output WCS.
single = boolean(default=False)
blendheaders = boolean(default=True)
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image.
in_memory = boolean(default=True)
good_bits = string(default='~DO_NOT_USE+NON_SCIENCE')  # The good bits to use␣
↪for building the resampling 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.

set_drizzle_defaults()

Set the default parameters for drizzle.

update_phot_keywords(*model*)

Update pixel scale keywords

Class Inheritance Diagram

Python Interface to Drizzle: ResampleData()

romancal.resample.resample Module

Classes

OutputTooLargeError	Raised when the output is too large for in-memory instantiation
ResampleData(input_models[, output, single, ...])	This is the controlling routine for the resampling process.

OutputTooLargeError

exception romancal.resample.resample.**OutputTooLargeError**

Raised when the output is too large for in-memory instantiation

ResampleData

class romancal.resample.resample.**ResampleData**(*input_models*, *output=None*, *single=False*, *blendheaders=True*, *pixfrac=1.0*, *kernel='square'*, *fillval='INDEF'*, *wht_type='ivm'*, *good_bits='0'*, *pscale_ratio=1.0*, *pscale=None*, ***kwargs*)

Bases: object

This is the controlling routine for the resampling process.

Notes

This routine performs the following operations:

1. Extracts parameter settings from input model, such as pixfrac, weight type, exposure time (if relevant), and kernel, and merges them with any user-provided values.
2. Creates output WCS based on input images and define mapping function between all input arrays and the output array. Alternatively, a custom, user-provided WCS object can be used instead.
3. Updates output data model with output arrays from drizzle, including 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*) – filename for output
- **kwargs** (*dict*) – Other parameters.

Note: `output_shape` is in the `x, y` order.

Note: `in_memory` controls whether or not the resampled array from `resample_many_to_many()` should be kept in memory or written out to disk and deleted from memory. Default value is `True` to keep all products in memory.

Methods Summary

do_drizzle()	Pick the correct drizzling mode based on `self.single`.
drizzle_arrays(insci, inwht, input_wcs, ...)	Low level routine for performing 'drizzle' operation on one image.
resample_exposure_time(output_model)	Resample the exposure time from `self.input_models` to the `output_model`.
resample_many_to_many()	Resample many inputs to many outputs where outputs have a common frame.
resample_many_to_one()	Resample and coadd many inputs to a single output.
resample_variance_array(name, output_model)	Resample variance arrays from `self.input_models` to the `output_model`.
update_exposure_times(output_model, exptime_tot)	Update exposure time metadata (in-place).

Methods Documentation

do_drizzle()

Pick the correct drizzling mode based on `self.single`.

static drizzle_arrays(*insci*, *inwht*, *input_wcs*, *output_wcs*, *outsci*, *outwht*, *outcon*, *uniqid=1*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *pixfrac=1.0*, *kernel='square'*, *fillval='INDEF'*, *wtscale=1.0*)

Low level routine for performing 'drizzle' operation on one image.

The interface is compatible with STScI code. All images are Python `ndarrays`, instead of filenames. File handling (input and output) is performed by the calling routine.

Parameters

- **insci** (*2d array*) – A 2d `numpy` array containing the input image to be drizzled.
- **inwht** (*2d array*) – A 2d `numpy` array containing the pixel by pixel weighting. Must have the same dimensions as insci. If none is supplied, the weighting is set to one.
- **input_wcs** (`gwcs.wcs.WCS` object) – The world coordinate system of the input image.
- **output_wcs** (`gwcs.wcs.WCS` object) – The world coordinate system of the output image.
- **outsci** (*2d array*) – A 2d `numpy` array containing the output image produced by drizzling. On the first call it should be set to zero. Subsequent calls it will hold the intermediate results. This is modified in-place.
- **outwht** (*2d array*) – A 2d `numpy` array containing the output counts. On the first call it should be set to zero. On subsequent calls it will hold the intermediate results. This is modified in-place.
- **outcon** (*2d or 3d array, optional*) – A 2d or 3d `numpy` array holding a bitmap of which image was an input for each output pixel. Should be integer zero on first call. Subsequent calls hold intermediate results. This is modified in-place.
- **uniqid** (*int, optional*) – The id number of the input image. Should be one the first time this function is called and incremented by one on each subsequent call.
- **xmin** (*float, None, optional*) – on the input image. Only pixels on the input image inside this rectangle will have their flux added to the output image. Xmin sets the minimum value of the x dimension. The x dimension is the dimension that varies quickest on the image. If the value is zero, no minimum will be set in the x dimension. All four parameters are zero based, counting starts at zero.
- **xmax** (*float, None, optional*) – Sets the maximum value of the x dimension on the bounding box of the input image. If the value is zero, no maximum will be set in the x dimension, the full x dimension of the output image is the bounding box.
- **ymin** (*float, None, optional*) – Sets the minimum value in the y dimension on the bounding box. The y dimension varies less rapidly than the x and represents the line index on the input image. If the value is zero, no minimum will be set in the y dimension.
- **ymax** (*float, None, optional*) – Sets the maximum value in the y dimension. If the value is zero, no maximum will be set in the y dimension, the full x dimension of the output image is the bounding box.
- **pixfrac** (*float, optional*) – The fraction of a pixel that the pixel flux is confined to. The default value of 1 has the pixel flux evenly spread across the image. A value of 0.5 confines it to half a pixel in the linear dimension, so the flux is confined to a quarter of the pixel area when the square kernel is used.

- **kernel** (*str, optional*) – The name of the kernel used to combine the input. The choice of kernel controls the distribution of flux over the kernel. The kernel names are: `'square'`, `'gaussian'`, `'point'`, `'tophat'`, `'turbo'`, `'lanczos2'`, and `'lanczos3'`. The `'square'` kernel is the default.
- **fillval** (*str, optional*) – The value a pixel is set to in the output if the input image does not overlap it. The default value of INDEF does not set a value.

Returns
A tuple with three values: a version string, the number of pixels on the input image that do not overlap the output image, and the number of complete lines on the input image that do not overlap the output input image.

Return type
tuple

resample_exposure_time(*output_model*)

Resample the exposure time from `self.input_models` to the `output_model`.

Create an exposure time image that is the drizzled sum of the input images.

resample_many_to_many()

Resample many inputs to many outputs where outputs have a common frame.

Coadd only different detectors of the same exposure (e.g. map SCA 1 and 10 onto the same output image), as they image different areas of the sky.

Used for outlier detection

resample_many_to_one()

Resample and coadd many inputs to a single output. Used for level 3 resampling

resample_variance_array(*name*, *output_model*)

Resample variance arrays from `self.input_models` to the `output_model`.

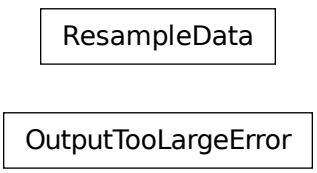
Resample the `name` variance array to the same name in `output_model`, using a cumulative sum.

This modifies `output_model` in-place.

update_exposure_times(*output_model*, *exptime_tot*)

Update exposure time metadata (in-place).

Class Inheritance Diagram



Resample Utilities

romancal.resample.resample_utils.**build_driz_weight**(*model*, *weight_type=None*, *good_bits: str = None*)

Builds the drizzle weight map for resampling.

Parameters

- **model** (*object*) – The input model.
- **weight_type** (*str, optional*) – The type of weight to use. Allowed values are 'ivm' or 'exptime'. Defaults to None.
- **good_bits** (*str, optional*) – The good bits to use for building the mask. Defaults to None.

Returns
The drizzle weight map.

Return type
numpy.ndarray

Raises
ValueError – If an invalid weight type is provided.

Examples

```
model = get_input_model()
weight_map = build_driz_weight(model, weight_type='ivm', good_bits=[1, 2, 3])
print(weight_map)
```

romancal.resample.resample_utils.**build_mask**(*dqarr*, *bitvalue*)

Build a bit mask from an input DQ array and a bitvalue flag.

Parameters

- **dqarr** (*numpy.ndarray*) – Input DQ array.
- **bitvalue** (*str*) – Bitvalue flag.

Returns
Bit mask where 1 represents good and 0 represents bad.

Return type
ndarray

Notes

- The function interprets the bitvalue flag using the `astropy.nddata.bitmask.interpret_bit_flags` function.
- If the bitvalue is None, the function returns a bit mask with all elements set to 1.
- Otherwise, the function performs a bitwise AND operation between the dqarr and the complement of the bitvalue, and then applies a logical NOT operation to obtain the bit mask.
- The resulting bit mask is returned as a `numpy.ndarray` of dtype `numpy.uint8`.

romancal.resample.resample_utils.**calc_gwcs_pixmap**(*in_wcs*, *out_wcs*, *shape=None*)

Generate a pixel map grid using the input and output WCS.

Parameters

- **in_wcs** (WCS) – Input WCS.
- **out_wcs** (WCS) – Output WCS.
- **shape** (*tuple, optional*) – Shape of the data. If provided, the bounding box will be calculated from the shape. If not provided, the bounding box will be calculated from the input WCS.

Returns

pixmap – The calculated pixel map grid.

Return type

ndarray

romancal.resample.resample_utils.**decode_context**(*context*, *x*, *y*)

Get 0-based indices of input images that contributed to (resampled) output pixel with coordinates `x` and `y`.

Parameters

- **context** (*numpy.ndarray*) – A 3D ndarray of integral data type.
- **x** (*int, list of integers, numpy.ndarray of integers*) – X-coordinate of pixels to decode (3rd index into the `context` array)
- **y** (*int, list of integers, numpy.ndarray of integers*) – Y-coordinate of pixels to decode (2nd index into the `context` array)

Returns

- A list of `numpy.ndarray` objects each containing indices of input images
- that have contributed to an output pixel with coordinates `x` and `y`.
- *The length of returned list is equal to the number of input coordinate*
- arrays `x` and `y`.

Examples

An example context array for an output image of array shape `(5, 6)` obtained by resampling 80 input images.

```
con = np.array(
    [[[0, 0, 0, 0, 0, 0],
      [0, 0, 0, 36196864, 0, 0],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 537920000, 0, 0, 0]],
     [[0, 0, 0, 0, 0, 0,],
      [0, 0, 0, 67125536, 0, 0],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 163856, 0, 0, 0]],
     [[0, 0, 0, 0, 0, 0],
      [0, 0, 0, 8203, 0, 0],
      [0, 0, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
      [0, 0, 0, 0, 0, 0],
      [0, 0, 32865, 0, 0, 0]]],
    dtype=np.int32
)
decode_context(con, [3, 2], [1, 4])
[array([ 9, 12, 14, 19, 21, 25, 37, 40, 46, 58, 64, 65, 67, 77]),
array([ 9, 20, 29, 36, 47, 49, 64, 69, 70, 79])]
```

romancal.resample.resample_utils.**make_output_wcs**(*input_models*, *pscale_ratio=None*, *pscale=None*, *rotation=None*, *shape=None*, *crpix: Tuple[float, float] = None*, *crval: Tuple[float, float] = None*)

Generate output WCS here based on footprints of all input WCS objects

Parameters

- **input_models** (list of roman_datamodels.datamodels.DataModel) – Each datamodel must have a gwcs.wcs.WCS object.
- **pscale_ratio** (*float, optional*) – Ratio of input to output pixel scale. Ignored when pscale is provided.
- **pscale** (*float, None, optional*) – Absolute pixel scale in degrees. When provided, overrides pscale_ratio.
- **rotation** (*float, None, optional*) – Position angle (in degrees) of output image's Y-axis relative to North. A value of 0.0 would orient the final output image to be North up. The default of None specifies that the images will not be rotated, but will instead be resampled in the default orientation for the camera with the x and y axes of the resampled image corresponding approximately to the detector axes.
- **shape** (*tuple of int, None, optional*) – Shape of the image (data array) using numpy.ndarray convention (ny first and nx second). This value will be assigned to pixel_shape and array_shape properties of the returned WCS object.
- **crpix** (*tuple of float, None, optional*) – Position of the reference pixel in the image array. If ref_pixel is not specified, it will be set to the center of the bounding box of the returned WCS object.
- **crval** (*tuple of float, None, optional*) – Right ascension and declination of the reference pixel. Automatically computed if not provided.

Returns

output_wcs – WCS object, with defined domain, covering entire set of input frames

Return type

object

romancal.resample.resample_utils.**reproject**(*wcs1*, *wcs2*)

Given two WCSs or transforms return a function which takes pixel coordinates in the first WCS or transform and computes them in the second one. It performs the forward transformation of wcs1 followed by the inverse of wcs2.

Parameters

- **wcs1** (astropy.wcs.WCS or gwcs.wcs.WCS or astropy.modeling.Model) – WCS objects.
- **wcs2** (astropy.wcs.WCS or gwcs.wcs.WCS or astropy.modeling.Model) – WCS objects.

Returns
Function to compute the transformations. It takes `(x, y)` positions in `wcs1` and returns `(x, y)` positions in `wcs2`.

Return type
func

romancal.resample Package

Classes

ResampleStep([name, parent, config_file, ...])	Resample input data onto a regular grid using the drizzle algorithm.

ResampleStep

class romancal.resample.**ResampleStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: *RomanStep*

Resample input data onto a regular grid using the drizzle algorithm.

Note: When supplied via `output_wcs`, a custom WCS overrides other custom WCS parameters such as `output_shape` (now computed from by `output_wcs.bounding_box`), `crpix`

Parameters
input (str, `roman_datamodels.datamodels.DataModel`, or *ModelContainer*) – If a string is provided, it should correspond to either a single ASDF filename or an association filename. Alternatively, a single DataModel instance can be provided instead of an ASDF filename. Multiple files can be processed via either an association file or wrapped by a *ModelContainer*.

Returns
A mosaic datamodel with the final output frame.

Return type
`roman_datamodels.datamodels.MosaicModel`

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

class_alias
reference_file_types
spec

Methods Summary

process(input)	This is where real work happens.
set_drizzle_defaults()	Set the default parameters for drizzle.
update_phot_keywords(model)	Update pixel scale keywords

Attributes Documentation

class_alias = 'resample'

reference_file_types: ClassVar = []

spec

```
pixfrac = float(default=1.0) # change back to None when drizpar reference files␣
↪are updated
kernel = string(default='square') # change back to None when drizpar reference␣
↪files are updated
fillval = string(default='INDEF' ) # change back to None when drizpar reference␣
↪files are updated
weight_type = option('ivm', 'exptime', None, default='ivm')  # change back to␣
↪None when drizpar ref update
output_shape = int_list(min=2, max=2, default=None)  # [x, y] order
crpix = float_list(min=2, max=2, default=None)
crval = float_list(min=2, max=2, default=None)
rotation = float(default=None)
pixel_scale_ratio = float(default=1.0) # Ratio of input to output pixel scale
pixel_scale = float(default=None) # Absolute pixel scale in arcsec
output_wcs = string(default='')  # Custom output WCS.
single = boolean(default=False)
blendheaders = boolean(default=True)
allowed_memory = float(default=None)  # Fraction of memory to use for the␣
↪combined image.
in_memory = boolean(default=True)
good_bits = string(default='~DO_NOT_USE+NON_SCIENCE')  # The good bits to use␣
↪for building the resampling 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.

set_drizzle_defaults()

Set the default parameters for drizzle.

update_phot_keywords(*model*)

Update pixel scale keywords

Class Inheritance Diagram

1.5 Reference Files

Many pipeline steps rely on the use of reference files that contain different types of calibration data or information necessary for processing the data. The reference files are instrument-specific and are periodically updated as the data processing evolves and the understanding of the instruments improves. They are created, tested, and validated by the Roman Instrument Team. They ensure all the files are in the correct format and have all required attributes. The files are then delivered to the Reference Data for Calibration and Tools (ReDCaT) Management Team. The result of this process is the files being ingested into the Roman Calibration Reference Data System (CRDS), 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 Information*, as well as in the documentation for each Calibration Step that uses a reference file.

1.5.1 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='roman_0017.pmap'
```

Within STScI, the current storage location for all Roman CRDS reference files is:

```
/grp/crds/roman
```

1.6 Parameters

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

```
$ strun roman_elp -h
$ strun romancal.dq_init.DQInitStep -h

$strun roman_hlp -h
$strun romancal.skymatch.SkyMatchStep -h
```

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

```
$ strun roman_elp r0008308002010007027_06311_0019_WFI01_uncal.asdf --save_calibrated_
↪ramp=true
```

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

```
$ strun roman_elp r0008308002010007027_06311_0019_WFI01_uncal.asdf
      --steps.dark_current.override_dark='my_dark.asdf'
```

1.6.1 Universal Parameters

The set of parameters that are common to all pipelines and steps are referred to as **universal parameters** and are described below.

Output Directory

By default, all pipeline and step outputs will drop into the current working directory, i.e., the directory in which the process is running. To change this, use the `output_dir` parameter. For example, to have all output from `roman_elp`, including any saved intermediate steps, appear in the sub-directory `calibrated`, use

```
$ strun roman_elp r0008308002010007027_06311_0019_WFI01_uncal.asdf
    --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 roman_elp r0008308002010007027_06311_0019_WFI01_uncal.asdf
    --output_dir=calibrated
    --steps.dark_current.output_file='dark_sub.asdf'
    --steps.dark_current.output_dir='dark_calibrated'
```

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.

1.7 I/O File Conventions

1.7.1 Input Files

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

Data files are the science input, such as exposure ASDF files. All files are assumed to be co-resident in the directory where the primary input file is located.

1.7.2 Output Files

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

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

- The name of the input file
- As specified by the *output_file* parameter

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

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

Individual Step Outputs

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

```
$ strun romancal.dq_init.DQInitStep r00083080020100007027_06311_0019_WFI01_uncal.asdf
```

produces an output file named `r00083080020100007027_06311_0019_WFI01_dq_init.asdf`.

Suffix Definitions

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

Product	Suffix
Uncalibrated raw input	_uncal
DQ initialization	_dqinit
Saturation detection	_saturation
Reference Pixel Correction	_refpix
Linearity correction	_linearity
Dark current	_darkcurrent
Corrected ramp data	_rampfit
Optional fitting results from ramp_fit step	_fitopt
Assign WCS	_assignwcs
Flat field	_flat
Photometric calibration	_photomstep
Source Detection	_sourcedetection
Tweakreg (Allign to GAIA)	_tweakregstep
Calibrated image	_cal

1.8 Error Propagation

1.8.1 Description

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

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

Step	Stage	Creates arrays	Updates arrays	Step uses
ramp_fitting	ELPP	VAR_POISSON, VAR_RNOISE	ERR	None
flat_field	ELPP	VAR_FLAT	ERR, VAR_POISSON, VAR_RNOISE	None
outlier_detection	HLPP	None	None	ERR

1.8.2 ELPP Processing

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

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

ramp_fitting

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

flat_field

The SCI array of the exposure being processed is divided by the flat-field reference image, and the VAR_POISSON and VAR_RNOISE arrays are divided by the square of the flat. A VAR_FLAT array is created, computed from the science data and the flat-field reference file ERR array. The science data ERR array is then updated to be the square root of the quadratic sum of the three variance arrays. For more details see *flat_field*.

1.8.3 HLPP Processing

HLPP pipelines perform additional instrument-level and observing-mode corrections and calibrations to produce fully calibrated exposures. The various steps that apply corrections and calibrations apply those same corrections/calibrations to all variance arrays and update the total ERR array.

outlier_detection

The `outlier_detection` step is used in all Stage 3 pipelines.

1.9 Data Products Information

1.9.1 Processing Levels and Product Stages

Here we describe the structure and content of the most frequently used forms of files for Roman science data products, which are in ASDF format. Each type of ASDF file is the result of serialization of a corresponding *DataModel*. All data models are defined by their schemas.

Within the various STScI internal data processing and archiving systems that are used for routine processing of Roman data, there are some different uses of terminology to refer to different levels of processing. The WFI data is converted into ASDF files by Science Data Formatting (SDF), level 0. SDF also inserts data from the engineering database and from the proposal database to create the level 1 files. SDF produces one ASDF file per detector and exposure and these level 1 files are used as input to the Exposure Level Processing. The output of the exposure level processing is a level 2 file.

Data Processing Levels	User Data Product Stages	MAST Product
Level 0, Science telemetry	Not available to users	Not available to users
Level 1, Uncalibrated files	Level 1 - ASDF file, fully populated by SDF	1 - Uncalibrated exposures
Exposure Level Processing	Level 2 - Fully calibrated exposures	2 - Calibrated exposures
High Level Processing	Level 3 - Combined data	3 - Combined data
High Level Processing	Level 4 - Catalogs, segmentation images	4 - Combined data

Level 1 and Level 2 products packaged the data from individual detectors in a single ASDF file. Level 3 are data resampled on a sky cell. The definitions of the sky cells is TBD. The definition of Level 3 and Level 4 products is being finalized.

1.9.2 File Naming Schemes

Exposure file names

The names of the exposure level data are constructed with information from the science header of the exposure, allowing users to map it to the observation in their corresponding APT files. The ASDF file naming scheme for the Exposure products is:

```
r<ppppp><cc><aaa><sss><ooo><vvv>_<gg><s><aa>_<eeee>_<detector>_<prodType>.asdf
```

where

- ppppp: program ID number
- cc: Execution Plan number
- aaa: Pass Number (within execution plan)
- sss: Segment Number (within pass)
- ooo: observation number
- vvv: visit number
- gg: visit group
- s: sequence ID (1=prime, >1 parallel)
- aa: activity number (base 36)
- eeee: exposure number
- detector: detector name (e.g. WFI01, WFI02, . . .)
- prodType: product type identifier (e.g. 'uncal', 'cal')

The standard <prodType> for the pipeline are uncal and cal, for the input products and resulting calibrated product. There are optional suffixes for intermediate products that are not routinely produced by the pipeline and are based of the processing level and can include dqinit, saturation, linearity, jump, darkcurrent, rampfit, assignwcs, flat, and photom.

An example Exposure Level 1 upcalibrated ASDF file name is:

```
r0000101001001001001_01101_0001_WFI01_uncal.asdf
```

An example Exposure Level 2 product ASDF file name is:

```
r0000101001001001001_01101_0001_WFI01_cal.asdf
```

1.9.3 Data Product Types

The following tables contain lists of all data product types.

Exposure Level Data Products

Pipeline	Input	Output(s)	Base	Units	Description
romancal.pipeline.ExposurePipeline	uncal	cal	Exp	DN/s	2-D calibrated data
romancal.pipeline.HighLevelPipeline	cal	i2d	Exp	DN/s	2-D calibrated data

Exposure Pipeline Steps And Data Products

The following table contains lists of all data product types for the Exposure pipeline, as given by their file name suffix. The input uncal file and the final cal file are the only files that are produced in the standard processing. All the other are optional (opt) files that can be produced when the user is running the pipeline. The input for each optional step is the output of the preceding step.

Pipeline Step	Input	Output suffix	Data Model	Units	Description
		uncal	ScienceRawModel	DN	3-D uncalibrated exposure data
dq_init	uncal	dqinit (opt)	RampModel	DN	3-D data quality flags applied
saturation		saturation (opt)	RampModel	DN	3-D data saturated values flagged
refpix		refpix (opt)	RampModel	DN	3-D ref pix corrected data
linearity		linearity (opt)	RampModel	DN	3-D linearity corrected data
dark_current		darkcurrent (opt)	RampModel	DN	3-D dark current subtracted data
ramp_fitting		rampfit (opt)	ImageModel	DN/s	2-D slope corrected data
assign_wcs		assignwcs (opt)	ImageModel	DN/s	2-D data with gwcs
flatfield		flat (opt)	ImageModel	DN/s	2-D QE corrected data
photom		photom (opt)	ImageModel	DN/s	Add phometric data to header
source_detection		sourcedetectionstep (opt)	ImageModel	DN/s	Sources identified in the data
tweakreg		tweakregstep (opt)	ImageModel	DN/s	WCS aligned with GAIA
		cal	ImageModel	DN/s	2-D calibrated exposure data

High Level Processing Steps And Data Products

The following table contain lists of all data product types for the HighLevel Processsing (HLP) Pipeline, as given by their file name suffix. The input to the HLP is an association file (in JSON format), the output is a combined image. All the other are optional (opt) files that can be produced when the user is running the pipeline. The input for each optional step is the output of the preceding step.

Pipeline Step	Input	Output suffix	Data Model	Units	Description
		asn			
sky_match	asn	skymatch (opt)	ModelContainer	MJy/sr	A list of _cal files
outlier_detection		outlier_detection_step (opt)	ModelContainer	MJy/sr	A list of _cal files
resample		resamplestep (opt)	ModelContainer	MJy/sr	A list of _cal files
		i2d	MosaicModel	MJy/sr	A 2D resampled image

1.9.4 Science Products

The following sections describe the format and contents of each of the Roman ASDF science products.

Uncalibrated raw data: `uncal`

Exposure raw data products are designated by a file name suffix of "uncal." These files usually contain only the raw detector pixel values from an exposure, with the addition of meta data associated with the exposure. The resultantdq array is an optional array used to flag missing data in the data Formatting process.

data array		Data Type	Units	Dimensions
data	Required	uint16	DN	nresultants x nrows x ncols
amp33	Required	uint16	DN	nresultants x 4096 x 128
resultantdq	Optional	uint8	N/A	nresultants x nrows x ncols

- data: 3-D data array containing the raw pixel values. The first two dimensions are equal to the size of the detector readout, with the data from multiple resultants stored along the 3rd axis.
- amp33: This is the reference output from a dedicated SCA Output that reads additional Reference Pixels on the SCA that are separate from the full-frame array read out by the Science Outputs. This Output is active in parallel with either the 32 Science Outputs or the 1 Guide Window Output.
- resultantdq: An array that flags the location of any missing data discovered in the data formating process.

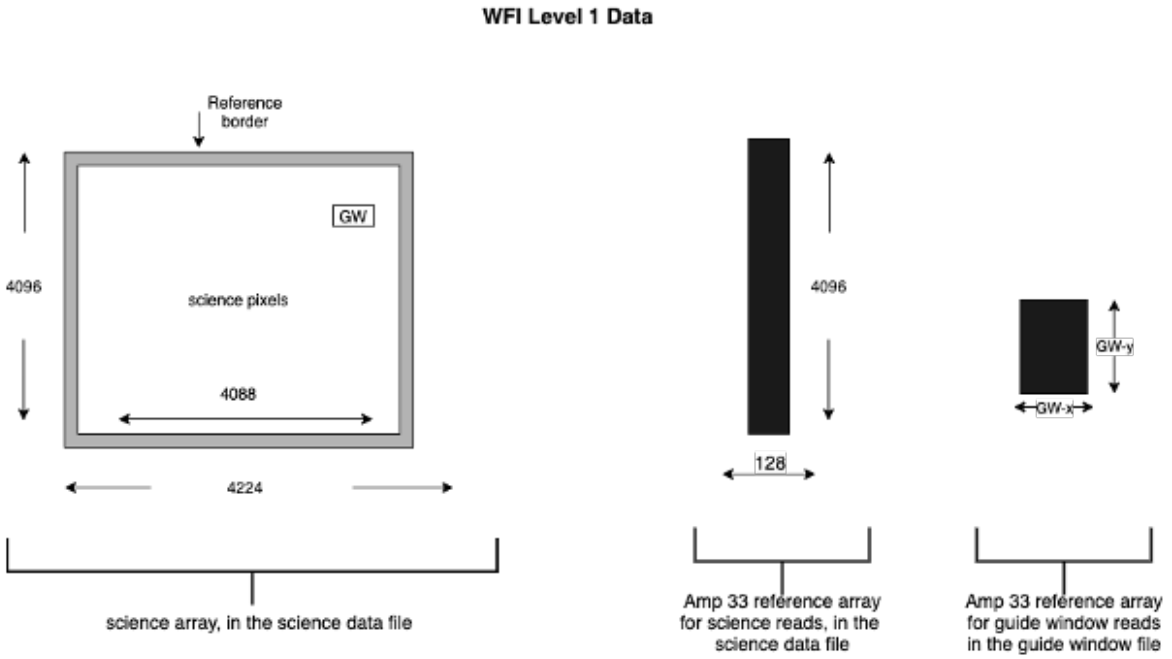
Ramp data: `ramp`

As raw data progress through the *romancal.pipeline.ExposurePipeline* pipeline they are stored internally in a `RampModel`. This type of data model is serialized to a `ramp` type ASDF file on disk. The original detector pixel values are converted from integer to floating-point data type. An ERR array and two types of data quality arrays are also added to the product. The ASDF file layout is as follows:

data array		Data Type	Units	Dimensions
data	Required	float32	DN	nresultants x nrows x ncols
pixeldq	Required	uint32	N/A	nrows x ncols
groupdq	Required	uint8	N/A	nresultants x nrows x ncols
err	Required	float32	DN	nresultants x nrows x ncols
amp33	Required	uint16	DN	nresultants x 4096 x 128
border_ref_pix_left	Required	float32	DN	nresultants x 4096 x 4
border_ref_pix_right	Required	float32	DN	nresultants x 4096 x 4
border_ref_pix_top	Required	float32	DN	nresultants x 4 x 4096
border_ref_pix_bottom	Required	float32	DN	nresultants x 4 x 4096

- data: 3-D data array containing the pixel values. The first two dimensions are equal to the size of the detector readout, with the data from multiple resultants stored along the 3rd axis.
- pixeldq: 2-D data array containing DQ flags that apply to all groups and all resultants for a given pixel (e.g. a hot pixel is hot in all groups and resultants).
- groupdq: 3-D data array containing DQ flags that pertain to individual reads within an exposure, such as the point at which a pixel becomes saturated within a given exposure.
- err: 3-D data array containing uncertainty estimates.
- amp33: Amp 33 reference pixel data.

- border_ref_pix_left: Copy of original border reference pixels, on left (from viewers perspective).
- border_ref_pix_right: Copy of original border reference pixels, on right (from viewers perspective).
- border_ref_pix_top: Copy of original border reference pixels, on the top (from viewers perspective).
- border_ref_pix_bottom: Copy of original border reference pixels, on the bottom (from viewers perspective).



Note: The reference pixels that are on the outer border of the science array are copied to these storage arrays (border_ref_pixel_<position>) at the dq_init step but are retained in the science array until being trimmed at the ramp fitting step.

Calibrated data: `cal`

The `cal` products are the result of runnng the *romancal.pipeline.ExposurePipeline* and yields an `The calibrated products are the result of an average over all integrations (``cal`)`.

data array		Data Type	Units	Dimensions
data	Required	float32	e^-/s	nrows x ncols
dq	Required	uint32	N/A	nrows x ncols
err	Required	float32	e^-/s	nrows x ncols
var_poisson	Required	float32	$(e^-/s)^2$	nrows x ncols
var_rnoise	Required	float32	$(e^-/s)^2$	nrows x ncols
var_flat	Required	float32	$(e^-/s)^2$	nrows x ncols
amp33	Required	uint16	DN	nresultants x 4096 x 128
border_ref_pix_left	Required	float32	DN	nresultants x 4096 x 4
border_ref_pix_right	Required	float32	DN	nresultants x 4096 x 4
border_ref_pix_top	Required	float32	DN	nresultants x 4 x 4096
border_ref_pix_bottom	Required	float32	DN	nresultants x 4 x 4096

- data: 2-D data array containing the calibrated pixel values.
- err: 2-D data array containing uncertainty estimates for each pixel. These values are based on the combined VAR_POISSON and VAR_RNOISE data (see below), given as standard deviation.
- dq: 2-D data array containing DQ flags for each pixel.
- var_poisson: 2-D data array containing the variance estimate for each pixel, based on Poisson noise only.
- var_rnoise: 2-D data array containing the variance estimate for each pixel, based on read noise only.
- var_flat: 2-D data array containing the variance estimate for each pixel, based on uncertainty in the flat-field.
- amp33: Amp 33 reference pixel data.
- border_ref_pix_left: Copy of original border reference pixels, on left (from viewers perspective).
- border_ref_pix_right: Copy of original border reference pixels, on right (from viewers perspective).
- border_ref_pix_top: Copy of original border reference pixels, on the top (from viewers perspective).
- border_ref_pix_bottom: Copy of original border reference pixels, on the bottom (from viewers perspective).

1.10 Associations

1.10.1 Association Overview

What are Associations?

Associations are basically just lists of things, mostly exposures, that are somehow related. With respect to Roman Mission 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 Roman

The basic chunk in which science data arrives from the observatory is termed an *exposure*. An exposure contains the data from detector reads that for the Roman mission are set by the MA table (Multiaccum Table). These resultants are the product transmitted to the ground and a set of these constitutes an exposure for the detector. 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 *Level 0*: The only transformation that has occurred is the extraction of the science data from the observatory telemetry into a ASDF file. At this point, the science exposures enter the calibration pipeline.

The pipeline consists of the ELP (Exposure Level Processing) and the HLP (High Level Processing) which together comprise three levels of data generation and processing: Level 1, Level 2, and Level 3. Level 1 data consist of uncalibrated individual exposures, containing raw pixel information, formatted into the shape of the detectors. Level 2 data have been processed to correct for instrument artifacts and have appropriate astrometric and geometric distortion information attached, and with the exception of grism data, are in units that have known scaling with flux. The resulting

files contain flux and spatially calibrated data, called *Level 2* data. The information contained in these files are still related to individual exposures.

In order to combine or compare exposures, the data are resampled to a regularized grid, removing the geometric distortion of the original pixels. This process creates *Level 3* data.

Utilities

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

1.10.2 Roman Associations

Roman Conventions

Association Naming

Note: Much of this is notional at the moment and will be refined in the upcoming meetings.

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

```
rPPPPP-TNNNN_YYYYMMDDtHHMMSS_ATYPE_MMM_asn.json
```

where:

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

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:

- `guider`: Intended for *dq_init* processing

Field Guide to File Names

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

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

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

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

- Files containing exposure-centric data

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

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

- Files containing target-centric data

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

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

Science Data Processing Workflow

General Workflow for Generating Association Products

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

1.10.3 Design

Association Design

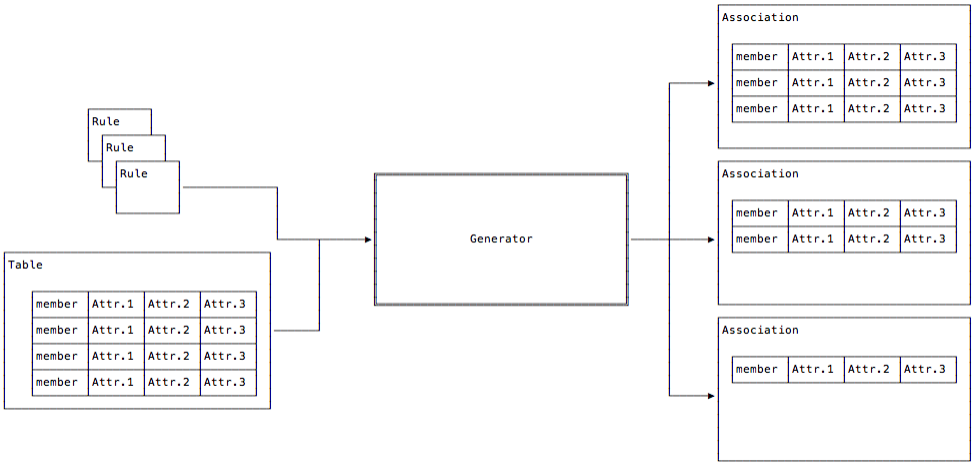


Fig. 1: Association Generator Overview

As introduced in the *Association Overview*, the figure above shows all the major players used in generating associations. Since this section will be describing the code design, the figure below is the overview but using the class names involved.

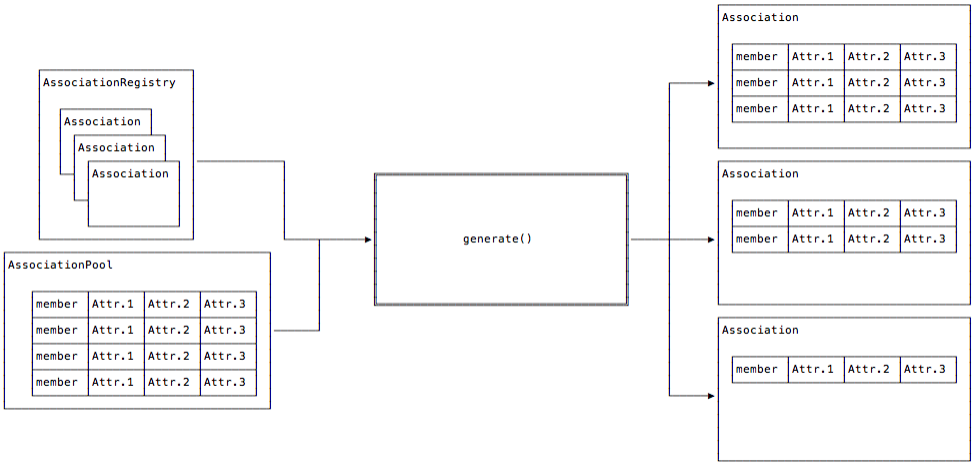


Fig. 2: Association Class Relationship overview

1.10.4 Reference

Association Commands

asn_from_list

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

Level2 Associations

Refer to TBD for a full description of Level2 associations.

To create a Level2 association, use the following command:

```
import romancal.associations.asn_from_list as asn_from_list
product_name = 'test_product'
items = {'r0000101001001001001_01101_0001_WFI01_uncal.asdf': 'science',
↪'r0000101001001001001_3_01101_0001_WFI01_uncal.asdf': 'guide_window', 'c':
↪'somethingelse'}
asn = asn_from_list.asn_from_list([(item, type_) for item, type_ in items.items()],␣
↪product_name=product_name, with_exptype=True)
asn['asn_rule']
'DMS_ELPP_Base'
```

an example product that has both a science and guide window exposures would look like the following:

```
asn['products']
[   {   'members': [   {   'expname': 'r0000101001001001001_01101_0001_WFI01_uncal.asdf',
                           'exptype': 'science'},
                       {   'expname': 'r0000101001001001001_3_01101_0001_WFI01_uncal.asdf
↪',
                           'exptype': 'guide_window'},
                       {'expname': 'c', 'exptype': 'somethingelse'}],
        'name': 'test_product'}]
```

To create a association with all the detectors for a given exposure from the command line,

```
asn_from_list -o detector_asn.json --product-name r0000101001001001001_01101_0001_WFI_
↪cal.asdf data/*_cal.asdf
```

where the individual calibrated detector files are in a data subdirectory.

1.11 Pipeline Modules

1.11.1 Pipeline Stages

The data from different observing modes needs to be processed with the proper pipeline levels listed below. The pipeline, and step selection in a pipeline, are usually based solely on the exposure type (`exposure.type` attribute). End-to-end calibration of Roman data is divided into levels of processing:

- The Exposure Level Processing (ELP) consists of detector-level corrections applied to each resultant, followed by ramp fitting. The output of the exposure level processing is a count rate image per exposure, that is aligned to the Gaia reference system. The details differ for imaging and spectroscopic exposures and can be found at *Exposure Level Processing*.
- The High Level Processing (HLP) uses overlapping exposures to match the sky background, detect aberrant data values and resample the image to produce a single undistorted product. Details are at: *High Level Image Processing*

  The table below represents the same information as described above, but alphabetically ordered by pipeline class.

Pipeline Class	Alias	Used For
ExposurePipeline	roman_elp	Exposure Level
HighLevelPipeline	roman_hlp	High Level

1.11.2 romancal.pipeline Package

This module collects all of the stpipe.Pipeline subclasses made available by this package.

Classes

ExposurePipeline(*args, **kwargs)	ExposurePipeline: Apply all calibration steps to raw Roman WFI ramps to produce a 2-D slope product.
HighLevelPipeline(*args, **kwargs)	HighLevelPipeline: Apply all calibration steps to the roman data to produce level 3 products.

ExposurePipeline

class romancal.pipeline.**ExposurePipeline**(**args*, ***kwargs*)

Bases: *RomanPipeline*

ExposurePipeline: Apply all calibration steps to raw Roman WFI ramps to produce a 2-D slope product. Included steps are: dq_init, saturation, linearity, dark current, jump detection, ramp_fit, assign_wcs, flatfield (only applied to WFI imaging data), photom, and source_detection.

See `Step.__init__` for the parameters.

Attributes Summary

class_alias
spec
step_defs

Methods Summary

create_fully_saturated_zeroed_image(input_r	Create zeroed-out image file
process(input)	Process the Roman WFI data
setup_output(input)	Determine the proper file name suffix to use later

Attributes Documentation

class_alias = 'roman_elp'

spec

```
save_calibrated_ramp = boolean(default=False)
save_results = boolean(default=False)
```

step_defs: ClassVar = {'assign_wcs': <class 'romancal.assign_wcs.assign_wcs_step.AssignWcsStep'>, 'dark_current': <class 'romancal.dark_current.dark_current_step.DarkCurrentStep'>, 'dq_init': <class 'romancal.dq_init.dq_init_step.DQInitStep'>, 'flatfield': <class 'romancal.flatfield.flat_field_step.FlatFieldStep'>, 'linearity': <class 'romancal.linearity.linearity_step.LinearityStep'>, 'photom': <class 'romancal.photom.photom_step.PhotomStep'>, 'rampfit': <class 'romancal.ramp_fitting.ramp_fit_step.RampFitStep'>, 'refpix': <class 'romancal.refpix.refpix_step.RefPixStep'>, 'saturation': <class 'romancal.saturation.saturation_step.SaturationStep'>, 'source_detection': <class 'romancal.source_detection.source_detection_step.SourceDetectionStep'>, 'tweakreg': <class 'romancal.tweakreg.tweakreg_step.TweakRegStep'>}

Methods Documentation

create_fully_saturated_zeroed_image(*input_model*)

Create zeroed-out image file

process(*input*)

Process the Roman WFI data

setup_output(*input*)

Determine the proper file name suffix to use later

HighLevelPipeline

class romancal.pipeline.**HighLevelPipeline**(**args*, ***kwargs*)

Bases: *RomanPipeline*

HighLevelPipeline: Apply all calibration steps to the roman data to produce level 3 products. Included steps are: `skymatch`, `outlier_detection` and `resample`.

See `Step.__init__` for the parameters.

Attributes Summary

class_alias
spec
step_defs

Methods Summary

process(input)	Process the Roman WFI data from Level 2 to Level 3

Attributes Documentation

class_alias = 'roman_hlp'

spec

```
save_results = boolean(default=False)
```

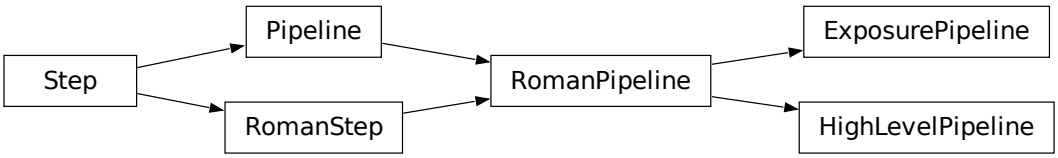
step_defs: ClassVar = {'flux': <class 'romancal.flux.flux_step.FluxStep'>, 'outlier_detection': <class 'romancal.outlier_detection.outlier_detection_step.OutlierDetectionStep'>, 'resample': <class 'romancal.resample.resample_step.ResampleStep'>, 'skymatch': <class 'romancal.skymatch.skymatch_step.SkyMatchStep'>}

Methods Documentation

process(*input*)

Process the Roman WFI data from Level 2 to Level 3

Class Inheritance Diagram



1.12 Reference File Information

1.12.1 Introduction

This document is intended to be a core reference guide to the formats, naming convention and data quality flags used by the reference files for pipeline steps requiring them, and is not intended to be a detailed description of each of those pipeline steps. It also does not give details on pipeline steps that do not use reference files. The present manual is the living document for the reference file specifications.

1.12.2 Reference File Naming Convention

Before reference files are ingested into CRDS, they are renamed following a convention used by the pipeline. As with any other changes undergone by the reference files, the previous names are kept in header, so the Instrument Teams can easily track which delivered file is being used by the pipeline in each step.

The naming of reference files uses the following syntax:

```
roman_<instrument>_<reftype>_<version>.<extension>
```

where

- `instrument` is currently "WFI"
- `reftype` is one of the type names listed in the table below
- `version` is a 4-digit version number (e.g. 0042)
- `extension` gives the file format, "asdf"

An example WFI FLAT reference file name would be "roman_wfi_flat_0042.asdf".

1.12.3 Reference File Types

Most reference files have a one-to-one relationship with calibration steps, e.g. there is one step that uses one type of reference file. Some steps, however, use several types of reference files and some reference file types are used by more than one step. The tables below show the correspondence between pipeline steps and reference file types. The first table is ordered by pipeline step, while the second is ordered by reference file type. Links to the reference file types provide detailed documentation on each reference file.

Pipeline Step	Reference File Type (reftype)
assign_wcs	DISTORTION
dark_current	*DARK*
dq_init	*MASK*
flatfield	*FLAT*
jump_detection	GAIN
	READNOISE
linearity	LINEARITY
photom	*PHOTOM*
ramp_fitting	GAIN
	READNOISE
saturation	*SATURATION*

Reference File Type (reftype)	Pipeline Step
DARK	*dark_current*
DISTORTION	*assign_wcs*
FLAT	*flatfield*
GAIN	*jump_detection*
	ramp_fitting
LINEARITY	*linearity*
MASK	*dq_init*
PHOTOM	*photom*
READNOISE	*jump_detection*
	ramp_fitting
SATURATION	*saturation*

1.12.4 Standard ASDF metadata

Al Roman science and reference files are ASDF files.

The required attributes Documenting Contents of Reference Files are:

Attribute	Comment
reftype	`FLAT Required values are listed in the discussion of each pipeline step.`
description	`Summary of file content and/or reason for delivery.`
author	`Fred Jones Person(s) who created the file.`
useafter	`YYYY-MM-DDThh:mm:ss Date and time after the reference files will be used. The T is required. Time string may NOT be omitted; use T00:00:00 if no meaningful value is available. Astropy Time objects are allowed.`
pedigree	`Options are 'SIMULATION' 'GROUND' 'DUMMY' 'INFLIGHT YYYY-MM-DD YYYY-MM-DD'`
history	`Description of Reference File Creation.`
telescope	`ROMAN Name of the telescope/project.`
instrument	`WFI Instrument name.`

1.12.5 Observing Mode Attributes

A pipeline module may require separate reference files for each instrument, detector, optical element, observation date, etc. The values of these parameters must be included in the reference file attributes. The observing-mode attributes are vital to the process of ingesting reference files into CRDS, as they are used to establish the mapping between observing modes and specific reference files. Some observing-mode attributes are also used in the pipeline processing steps.

The Keywords Documenting the Observing Mode are:

Keyword	Sample Value	Comment
detector	WFI01	Allowed values WFI01, WFI02, … WFI18
optical element	F158	Name of the filter element and includes PRISM and GRISM
exposure type	WFI_IMAGE	Allowed values WFI_IMAGE, WFI_GRATING, WFI_PRISM, WFI_DARK, WFI_FLAT, WFI_WFSC

Tracking Pipeline Progress

As each pipeline step is applied to a science data product, it will record a status indicator in a cal_step attribute of the science data product. These statuses may be included in the primary header of reference files, in order to maintain a history of the data that went into creating the reference file. Allowed values for the status Attribute are 'INCOMPLETE', 'COMPLETE' and 'SKIPPED'. The default value is set to 'INCOMPLETE'. The pipeline modules will set the value to 'COMPLETE' or 'SKIPPED'. If the pipeline steps are run manually and you skip a step the cal_step will remain 'INCOMPLETE'.

1.12.6 Data Quality Flags

Within science data files, the PIXELDQ flags are stored as 32-bit integers; the GROUPDQ flags are 8-bit integers. All calibrated data from a particular instrument and observing mode have the same set of DQ flags in the same (bit) order. The table below lists the allowed DQ flags. Only the first eight entries in the table below are relevant to the GROUPDQ array.

Flags for the DQ, PIXELDQ, and GROUPDQ Arrays.

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	GW_AFFECTED_DATA	Data affected by the GW read window
5	32	PERSISTENCE	High persistence (was RESERVED_2)
6	64	AD_FLOOR	Below A/D floor (0 DN, was RESERVED_3)
7	128	OUTLIER	Detected as outlier in coadded image
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

continues on next page

Table 2 – continued from previous page

Bit	Value	Name	Description
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	RESERVED_5	
27	134217728	RESERVED_6	
28	268435456	UNRELIABLE_RESET	Sensitive to reset anomaly
29	536870912	RESERVED_7	
30	1073741824	OTHER_BAD_PIXEL	A catch-all flag
31	2147483648	REFERENCE_PIXEL	Pixel is a reference pixel

1.12.7 Parameter Specification

There are a number of steps, such as *OutlierDetectionStep*, that define what data quality flags a pixel is allowed to have to be considered in calculations. Such parameters can be set in a number of ways.

First, the flag can be defined as the integer sum of all the DQ bit values from the input images DQ arrays that should be considered "good". For example, if pixels in the DQ array can have combinations of 1, 2, 4, and 8 and one wants to consider DQ flags 2 and 4 as being acceptable for computations, then the parameter value should be set to "6" (2+4). In this case a pixel having DQ values 2, 4, or 6 will be considered a good pixel, while a pixel with a DQ value, e.g., 1+2=3, 4+8="12", etc. will be flagged as a "bad" pixel.

Alternatively, one can enter a comma-separated or '+' separated list of integer bit flags that should be summed to obtain the final "good" bits. For example, both "4,8" and "4+8" are equivalent to a setting of "12".

Finally, instead of integers, the Roman mnemonics, as defined above, may be used. For example, all the following specifications are equivalent:

```
"12" == "4+8" == "4, 8" == "JUMP_DET, DROPOUT"
```

Note:

- The default value (0) will make *all* non-zero pixels in the DQ mask be considered "bad" pixels and the corresponding pixels will not be used in computations.
- Setting to `None` will turn off the use of the DQ array for computations.
- In order to reverse the meaning of the flags from indicating values of the "good" DQ flags to indicating the "bad" DQ flags, prepend '~' to the string value. For example, in order to exclude pixels with DQ flags 4 and 8 for computations and to consider as "good" all other pixels (regardless of their DQ flag), use a value of `~4+8`, or `~4,8`. A string value of `~0` would be equivalent to a setting of `None`.

1.13 STPIPE

1.13.1 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:

- Instantiating the Step directly from Python
- Reading the input from a parameter file

Running a Step from a parameter file

A parameter file contains one or more of a `Step`'s parameters. Any parameter not specified in the file will take its value from the defaults coded directly into the `Step`. Note that any parameter specified on the command line overrides all other values.

The format of parameter files is the *ASDF Parameter Files* format. Refer to the *minimal example* for a complete description of the contents. The rest of this document will focus on the step parameters themselves.

Every parameter file must contain the key `class`, followed by the optional `name` followed by any parameters that are specific to the step being run.

`class` specifies the Python class to run. It should be a fully-qualified Python path to the class. Step classes can ship with `stpipe` itself, they may be part of other Python packages, or they exist in freestanding modules alongside the configuration file.

`name` defines the name of the step. This is distinct from the class of the step, since the same class of Step may be configured in different ways, and it is useful to be able to have a way of distinguishing between them. For example, when Steps are combined into *Pipelines*, a Pipeline may use the same Step class multiple times, each with different configuration parameters.

The parameters specific to the Step all reside under the key `parameters`. The set of accepted parameters is defined in the Step's spec member. The easiest way to get started on a parameter file is to call `Step.export_config` and then edit the file that is created. This will generate an ASDF config file that includes every available parameter, which can then be trimmed to the parameters that require customization.

Here is an example parameter file (`do_cleanup.asdf`) that runs the (imaginary) step `stpipe.cleanup` to clean up an image.

```
#ASDF 1.0.0
#ASDF_STANDARD 1.3.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
class: stpipe.cleanup
name: MyCleanup
parameters:
  threshold: 42.0
  scale: 0.01
...
```

Running a Step from the commandline

The `strun` command can be used to run Steps from the commandline.

The first argument may be either:

- The a parameter file
- A Python class

Additional parameters may be passed on the commandline. These parameters override any defaults. Any extra positional parameters on the commandline are passed to the step's process method. This will often be input filenames.

To display a list of the parameters that are accepted for a given Step class, pass the `-h` parameter, and the name of a Step class or parameter file:

```
$ strun -h romancal.dq_init.DQInitStep
usage: strun [-h] [--logcfg LOGCFG] [--verbose] [--debug] [--save-parameters SAVE_
↪PARAMETERS] [--disable-crds-steppars]
            [--pre_hooks] [--post_hooks] [--output_file] [--output_dir] [--output_ext]␣
↪[--output_use_model] [--output_use_index]
            [--save_results] [--skip] [--suffix] [--search_output_file] [--input_dir] [-
↪-override_mask]
            cfg_file_or_class [args ...]

(selected) optional arguments:
  -h, --help       show this help message and exit
  --logcfg LOGCFG  The logging configuration file to load
  --verbose, -v    Turn on all logging messages
  --debug            When an exception occurs, invoke the Python debugger, pdb
  --save-parameters SAVE_PARAMETERS Save step parameters to specified file.
  --disable-crds-steppars Disable retrieval of step parameter references files from CRDS
  --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.asdf` is output to `foo_cleanup.asdf`.

Finally, the parameters a `Step` actually ran with can be saved to a new parameter file using the `--save-parameters` option. This file will have all the parameters, specific to the step, and the final values used.

Parameter Precedence

There are a number of places where the value of a parameter can be specified. The order of precedence, from most to least significant, for parameter value assignment is as follows:

1. Value specified on the command-line: `strun step.asdf --par=value_that_will_be_used`
2. Value found in the user-specified parameter file
3. `Step`-coded default, determined by the parameter definition `Step.spec`

For pipelines, if a pipeline parameter file specifies a value for a step in the pipeline, that takes precedence over any step-specific value found from a step-specific parameter file. The full order of precedence for a pipeline and its sub steps is as follows:

1. Value specified on the command-line: `strun pipeline.asdf --steps.step.par=value_that_will_be_used`
2. Value found in the user-specified pipeline parameter file: `strun pipeline.asdf`

3. Value found in the parameter file specified in a pipeline parameter file
4. `Pipeline`-coded default for itself and all sub-steps
5. `Step`-coded default for each sub-step

Debugging

To output all logging output from the step, add the `--verbose` option to the commandline. (If more fine-grained control over logging is required, see *Logging*).

To start the Python debugger if the step itself raises an exception, pass the `--debug` option to the commandline.

Running a Step in Python

There are a number of methods to run a step within a Python interpreter, depending on how much control one needs.

Step.from_cmdline()

For individuals who are used to using the `strun` command, `Step.from_cmdline` is the most direct method of executing a step or pipeline. The only argument is a list of strings, representing the command line arguments one would have used for `strun`. The call signature is:

```
Step.from_cmdline([string,...])
```

For example, given the following command-line:

```
$ strun romancal.pipeline.ExposurePipeline r0000101001001001001_01101_0001_WFI01_uncal.
↪asdf \
       --steps.jump.override_gain=roman_wfi_gain_0033.asdf
```

the equivalent `from_cmdline` call would be:

```
from romancal.pipeline import ExposurePipeline
ExposurePipeline.from_cmdline([' r0000101001001001001_01101_0001_WFI01_uncal.asdf',
                               'steps.jump.override_gain', 'roman_wfi_gain_0033.asdf'])
```

call()

Class method `Step.call` is the slightly more programmatic, and preferred, method of executing a step or pipeline. When using `call`, one gets the full configuration initialization that one gets with the `strun` command or `Step.from_cmdline` method. The call signature is:

```
Step.call(input, logcfg=None, **parameters)
```

The positional argument `input` is the data to be operated on, usually a string representing a file path or a *DataModel*. The optional keyword argument `config_file` is used to specify a local parameter file. The optional keyword argument `logcfg` is used to specify a logging configuration file. Finally, the remaining optional keyword arguments are the parameters that the particular step accepts. The method returns the result of the step. A basic example is:

```
from romancal.jump import JumpStep
output = JumpStep.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf')
```

makes a new instance of `JumpStep` and executes using the specified exposure file. `JumpStep` has a parameter `rejection_threshold`. To use a different value than the default, the statement would be:

```
output = JumpStep.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf',
                       rejection_threshold=42.0)
```

If one wishes to use a *parameter file*, specify the path to it using the `config_file` argument:

```
output = JumpStep.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf',
                       config_file='my_jumpstep_config.asdf')
```

run()

The instance method `Step.run()` is the lowest-level method to executing a step or pipeline. Initialization and parameter settings are left up to the user. An example is:

```
from romancal.flatfield import FlatFieldStep

mystep = FlatFieldStep()
mystep.override_sflat = 'sflat.asdf'
output = mystep.run(input)
```

`input` in this case can be a asdf file containing the appropriate data, or the output of a previously run step/pipeline, which is an instance of a particular *datamodel*.

Unlike the `call` class method, there is no parameter initialization that occurs, either by a local parameter file or from a CRDS-retrieved parameter reference file. Parameters can be set individually on the instance, as is shown above. Parameters can also be specified as keyword arguments when instantiating the step. The previous example could be re-written as:

```
from romancal.flatfield import FlatFieldStep

mystep = FlatFieldStep(override_sflat='sflat.asdf')
output = mystep.run(input)
```

Using the `.run()` method is the same as calling the instance directly. They are equivalent:

```
output = mystep(input)
```

Pipelines

It is important to note that a Pipeline is also a Step, so everything that applies to a Step in the *For Users* chapter also applies to Pipelines.

Configuring a Pipeline

This section describes how to set parameters on the individual steps in a pipeline. To change the order of steps in a pipeline, one must write a Pipeline subclass in Python. That is described in the *Pipelines* section of the developer documentation.

Just as with Steps, Pipelines can by configured either by a parameter file or directly from Python.

From a parameter file

A Pipeline parameter file follows the same format as a Step parameter file: *ASDF Parameter Files*

Here is an example pipeline parameter file for the `ExposurePipeline` class:

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://
↪github.com/asdf-format/asdf',
name: asdf, version: 2.13.0}
history:
extensions:
- !core/extension_metadata-1.0.0
 extension_class: asdf.extension.BuiltinExtension
 software: !core/software-1.0.0 {name: asdf, version: 2.13.0}
class: romancal.pipeline.exposure_pipeline.ExposurePipeline
meta:
author: <SPECIFY>
date: '2022-09-15T13:59:54'
description: Parameters for calibration step romancal.pipeline.exposure_pipeline.
↪ExposurePipeline
instrument: {name: <SPECIFY>}
origin: <SPECIFY>
pedigree: <SPECIFY>
reftype: <SPECIFY>
telescope: <SPECIFY>
useafter: <SPECIFY>
name: ExposurePipeline
parameters:
input_dir: ''
output_dir: null
output_ext: .asdf
output_file: null
output_use_index: true
output_use_model: false
post_hooks: []
pre_hooks: []
save_calibrated_ramp: false
save_results: true
search_output_file: true
skip: false
```

(continues on next page)

(continued from previous page)

```
suffix: null
steps:
- class: romancal.dq_init.dq_init_step.DQInitStep
name: dq_init
parameters:
 input_dir: ''
 output_dir: null
 output_ext: .asdf
 output_file: null
 output_use_index: true
 output_use_model: false
 post_hooks: []
 pre_hooks: []
save_results: false
search_output_file: true
skip: false
suffix: null
- class: romancal.saturation.saturation_step.SaturationStep
...
```

Just like a `Step`, it must have `name` and `class` values. Here the `class` must refer to a subclass of `stpipe.Pipeline`.

Following `name` and `class` is the `steps` section. Under this section is a subsection for each step in the pipeline. The easiest way to get started on a parameter file is to call `Step.export_config` and then edit the file that is created. This will generate an ASDF config file that includes every available parameter, which can then be trimmed to the parameters that require customization.

For each Step's section, the parameters for that step may either be specified inline, or specified by referencing an external parameter file just for that step. For example, a pipeline parameter file that contains:

```
- class: romancal.jump.jump_step.JumpStep
  name: jump
  parameters:
    flag_4_neighbors: true
    four_group_rejection_threshold: 190.0
    input_dir: ''
    max_jump_to_flag_neighbors: 1000.0
    maximum_cores: none
    min_jump_to_flag_neighbors: 10.0
```

is equivalent to:

```
steps:
- class: romancal.jump.jump_step.JumpStep
  name: jump
  parameters:
     config_file = myjump.asdf
```

with the file `myjump.asdf.` in the same directory:

```
class: romancal.jump.jump_step.JumpStep
name: jump
parameters:
```

(continues on next page)

(continued from previous page)

```
  flag_4_neighbors: true
  four_group_rejection_threshold: 190.0
```

If both a `config_file` and additional parameters are specified, the `config_file` is loaded, and then the local parameters override them.

Any optional parameters for each Step may be omitted, in which case defaults will be used.

From Python

A pipeline may be configured from Python by passing a nested dictionary of parameters to the Pipeline's constructor. Each key is the name of a step, and the value is another dictionary containing parameters for that step. For example, the following is the equivalent of the parameter file above:

```
from stpipe.pipeline import Image2Pipeline

steps = {
    'jump':{'rejection_threshold': 180.,
            'three_group_rejection_threshold': 190.,
            'four_group_rejection_threshold':195.
}

pipe = ExposurePipeline(steps=steps)
```

Running a Pipeline

From the commandline

The same `strun` script used to run Steps from the commandline can also run Pipelines.

The only wrinkle is that any parameters overridden from the commandline use dot notation to specify the parameter name. For example, to override the `rejection_threshold` value on the `jump` step in the example above, one can do:

```
> strun romancal.pipeline.ExposurePipeline --steps.jump.rejection_threshold=180.
```

From Python

Once the pipeline has been configured (as above), just call the instance to run it.

```
pipe(input_data)
```

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 ASDF models that are the result of a step are saved as standalone ASDF files to make them more easily used by external tools. The filenames are based on the name of the substep within the pipeline.

Hooks

Each Step in a pipeline can also have pre- and post-hooks associated. Hooks themselves are Step instances, but there are some conveniences provided to make them easier to specify in a parameter file.

Pre-hooks are run right before the Step. The inputs to the pre-hook are the same as the inputs to their parent Step. Post-hooks are run right after the Step. The inputs to the post-hook are the return value(s) from the parent Step. The return values are always passed as a list. If the return value from the parent Step is a single item, a list of this single item is passed to the post hooks. This allows the post hooks to modify the return results, if necessary.

Hooks are specified using the `pre_hooks` and `post_hooks` parameters associated with each step. More than one pre- or post-hook may be assigned, and they are run in the order they are given. There can also be `pre_hooks` and `post_hooks` on the Pipeline as a whole (since a Pipeline is also a Step). Each of these parameters is a list of strings, where each entry is one of:

- An external commandline application. The arguments can be accessed using {0}, {1} etc. (See `stpipe.subproc.SystemCall`).
- A dot-separated path to a Python Step class.
- A dot-separated path to a Python function.

Logging

Log messages are emitted from each Step at different levels of importance. The levels used are the standard ones for Python (from least important to most important:

- 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.
- The file specified with the `logcfg` keyword to a .call() execution of a Step or Pipeline.
- 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 section of the Python standard library.

Examples

The following configuration turns on all log messages and saves them in the file myrun.log:

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

In a scenario where the user is debugging a particular Step, they may want to hide all logging messages except for that Step, and stop when hitting any warning for that Step:

```
[*]
level = CRITICAL

[*.MyStep]
level = INFO
break_level = WARNING
```

ASDF Parameter Files

ASDF is the format of choice for parameter files. ASDF stands for "Advanced Scientific Data Format", a general purpose, non-proprietary, and system-agnostic format for the dissemination of data. Built on YAML, the most basic file is text-based requiring minimal formatting.

To create a parameter file, the most direct way is to choose the Pipeline class, Step class, or already existing .asdf or .cfg file, and run that step using the `--save-parameters` option. For example, to get the parameters for the `ExposurePipeline` pipeline, do the following:

```
$ strun --save-parameters=exp_pars.asdf roman_elp r0000101001001001001_01101_0001_WFI01_
↪uncal.asdf
```

Once created and modified as necessary, the file can now be used by `strun` to run the step/pipeline with the desired parameters:

```
$ strun exp_pars.asdf r0000101001001001001_01101_0001_WFI01_uncal.asdf
```

The remaining sections will describe the file format and contents.

File Contents

To describe the contents of an ASDF file, the configuration for the step `roman_elp` will be used as the example:

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://
↪github.com/asdf-format/asdf',
  name: asdf, version: 2.13.0}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.13.0}
class: romancal.pipeline.exposure_pipeline.ExposurePipeline
meta:
  author: <SPECIFY>
  date: '2022-09-15T13:59:54'
  description: Parameters for calibration step romancal.pipeline.exposure_pipeline.
↪ExposurePipeline
  instrument: {name: <SPECIFY>}
  origin: <SPECIFY>
  pedigree: <SPECIFY>
  reftype: <SPECIFY>
  telescope: <SPECIFY>
  useafter: <SPECIFY>
name: ExposurePipeline
parameters:
  input_dir: ''
  output_dir: null
```

(continues on next page)

(continued from previous page)

```
  output_ext: .asdf
  output_file: null
  output_use_index: true
  output_use_model: false
  post_hooks: []
  pre_hooks: []
  save_calibrated_ramp: false
  save_results: true
  search_output_file: true
  skip: false
  suffix: null
steps:
- class: romancal.jump.jump_step.JumpStep
  name: jump
  parameters:
    flag_4_neighbors: true
    four_group_rejection_threshold: 190.0
    input_dir: ''
    max_jump_to_flag_neighbors: 1000.0
    maximum_cores: none
    min_jump_to_flag_neighbors: 10.0
    output_dir: null
    output_ext: .asdf
    output_file: null
    output_use_index: true
    output_use_model: false
    post_hooks: []
    pre_hooks: []
    rejection_threshold: 180.0
    save_results: false
    search_output_file: true
    skip: false
    suffix: null
    three_group_rejection_threshold: 185.0
...
```

Required Components

Preamble

The first 5 lines, up to and including the "—" line, define the file as an ASDF file. The rest of the file is formatted as one would format YAML data. Being YAML, the last line, containing the three `...` is essential.

class and name

There are two required keys at the top level: `class` and `parameters`. `parameters` is discussed below.

`class` specifies the Python class to run. It should be a fully-qualified Python path to the class. Step classes can ship with `stpipe` itself, they may be part of other Python packages, or they exist in freestanding modules alongside the configuration file. For example, to use the `SystemCall` step included with `stpipe`, set `class` to `stpipe.subprocess.SystemCall`. To use a class called `Custom` defined in a file `mysteps.py` in the same directory as the configuration file, set `class` to `mysteps.Custom`.

`name` defines the name of the step. This is distinct from the class of the step, since the same class of Step may be configured in different ways, and it is useful to be able to have a way of distinguishing between them. For example, when Steps are combined into *Pipelines*, a Pipeline may use the same Step class multiple times, each with different configuration parameters.

Parameters

`parameters` contains all the parameters to pass onto the step. The order of the parameters does not matter. It is not necessary to specify all parameters either. If not defined, the default, as defined in the code or values from CRDS parameter references, will be used.

Formatting

YAML has two ways of formatting a list of key/value pairs. In the above example, each key/value pair is on separate line. The other way is using a form that is similar to a Python `dict`. For example, the `parameters` block above could also have been formatted as:

```
parameters: {flag_4_neighbors: true, four_group_rejection_threshold: 190.0,
  max_jump_to_flag_neighbors: 1000.0, maximum_cores: none,
  min_jump_to_flag_neighbors: 10.0, output_dir: null, output_ext: .asdf,
  output_file: null, output_use_index: true, output_use_model: false,
  rejection_threshold: 180.0, three_group_rejection_threshold: 185.0}
```

Optional Components

The `asdf_library` and `history` blocks are necessary only when a parameter file is to be used as a parameter reference file in CRDS which is not currently implemented in the Roman pipeline.

Completeness

For any parameter file, it is not necessary to specify all step/pipeline parameters. Any parameter left unspecified will get, at least, the default value define in the step's code. If a parameter is defined without a default value, and the parameter is never assigned a value, an error will be produced when the step is executed.

Remember that parameter values can come from numerous sources. Refer to *Parameter Precedence* for a full listing of how parameters can be set.

From the `JumpStep` example, if all that needed to change is the `rejection_threshold` parameter with a setting of `80.0`, the `parameters` block need only contain the following:

```
parameters:
  rejection_threshold: 80.0
```

Pipeline Configuration

Pipelines are essentially steps that refer to sub-steps. As in the original cfg format, parameters for sub-steps can also be specified. All sub-step parameters appear in a key called `steps`. Sub-step parameters are specified by using the sub-step name as the key, then underneath and indented, the parameters to change for that sub-step. For example, to define the `rejection_threshold` of the `JumpStep` step in a `ExposurePipeline` parameter file, the parameter block would look as follows:

```
class: romancal.pipeline.exposure_pipeline.ExposurePipeline
parameters: {}
steps:
- class: romancal.jump.jump_step.JumpStep
  parameters:
    rejection_threshold: 80.0
```

As with step parameter files, not all sub-steps need to be specified. If left unspecified, the sub-steps will be run with their default parameter sets. For the example above, the other steps of `ExposurePipeline`, such as `assign_wcs` and `photom` would still be executed.

Similarly, to skip a particular step, one would specify `skip: true` for that substep. Continuing from the above example, to skip the `flatfield` step, the parameter file would look like:

```
class: romancal.pipeline.exposure_pipeline.ExposurePipeline
parameters: {}
steps:
- class: romancal.flatfield.flat_field_step.FlatFieldStep
  name: flatfield
  parameters:
    skip: true
```

Note: In the previous examples, one may have noted the line parameters: {}. Often when configuring a pipeline, one needs not set any parameters for the pipeline itself. However, the keyword `parameters` is required. As such, the value for `parameters` is defined as an empty dictionary, {}.

Python API

There are a number of ways to create an ASDF parameter file. From the command line utility `strun`, the option `--save-parameters` can be used.

Within a Python script, the method `Step.export_config(filename: str)` can be used. For example, to create a parameter file for `JumpStep`, use the following:

```
>>> from romancal.jump import JumpStep
>>> step = JumpStep()
>>> step.export_config('jump_step.asdf')
```

History

Parameter reference files also require at least one history entry. This can be found in the `history` block under `entries`:

```
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.13.0}
history:
  entries:
  - !core/history_entry-1.0.0 {description: Base values, time: !!timestamp '2019-10-29
      21:20:50'}
```

It is highly suggested to use the ASDF API to add history entries:

```
>>> import asdf
>>> cfg = asdf.open('config.asdf')
    #
    # Modify `parameters` and `meta` as necessary.
    #
>>> cfg.add_history_entry('Parameters modified for some reason')
>>> cfg.write_to('config_modified.asdf')
```

Roman, Parameters and Parameter References

In general, the default parameters for any pipeline or step are valid for nearly all instruments and observing modes. This means that when a pipeline or step is run without any explicit parameter setting, that pipeline or step will usually do the desired operation. Hence, most of the time there is no need for a parameter reference to be provided by the user. Only for a small set of observing mode/step combinations, will there be need to create a parameter reference. Even then, nearly all cases will involve changing a subset of a pipeline or step parameters.

Keeping this sparse-population philosophy in mind, for most parameter references, only those parameters that are explicitly changed should be specified in the reference. If adhered to, when a pipeline/step default value for a particular parameter needs to change, the change will be immediately available. Otherwise, all references that mistakenly set said parameter will need to be updated. See *Completeness* for more information.

Furthermore, every pipeline/step have a common set of parameters, listed below. These parameters generally affect the infrastructure operation of pipelines/steps, and should not be included in a parameter reference.

- input_dir
- output_ext
- output_use_index
- output_use_model
- post_hooks
- pre_hooks
- save_results
- search_output_file

Executing a pipeline or pipeline step via call()

The `call` method will create an instance and run a pipeline or pipeline step in a single call.

```
from romancal.pipeline import ExposurePipeline
result = ExposurePipeline.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf')

from romancal.linearity import LinearityStep
result = LinearityStep.call('r0000101001001001001_01101_0001_WFI01_dqinit.asdf')
```

To set custom parameter values when using the `call` method, set the parameters in the pipeline or parameter file and then supply the file using the `config_file` keyword:

```
# Calling a pipeline
result = ExposurePipeline.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf',␣
↪config_file='exp_pars.asdf'))

# Calling a step
result = LinearityStep.call('r0000101001001001001_01101_0001_WFI01_dqinit.asdf', config_
↪file = 'linearity_pars.asdf')
```

When running a pipeline, parameter values can also be supplied in the call to `call` itself by using a nested dictionary of step and parameter names:

```
result = ExposurePipeline.call('r0000101001001001001_01101_0001_WFI01_uncal.asdf',␣
↪config_file='exp_pars.asdf', steps={"jump":{"rejection_threshold": 200}})
```

When running a single step with `call`, parameter values can be supplied more simply:

```
result = JumpStep.call("r0000101001001001001_01101_0001_WFI01_dqinit.asdf", rejection_
↪threshold=200)
```

Running steps and pipelines with `call` also allows for the specification of a logging configuration file using the keyword `logcfg`:

```
result = ExposurePipeline.call("r0000101001001001001_01101_0001_WFI01_dqinit.asdf",
                               config_file="exp_pars.asdf",
                               logcfg="my-logging-config.cfg")
```

Where are the results?

A fundamental difference between running steps and pipelines in Python as opposed to from the command line using `strun` is whether files are created or not. When using `strun`, results are automatically saved to files because that is the only way to access the results.

However, when running within a Python interpreter or script, the presumption is that results will be used within the script. As such, results are not automatically saved to files. It is left to the user to decide when to save.

If one wishes for results to be saved by a particular `call`, use the parameter `save_results=True`:

```
result = JumpStep.call("r0000101001001001001_01101_0001_WFI01_dqinit.asdf",
                       rejection_threshold=200, save_results=True)
```

If one wishes to specify a different file name, rather than a system-generated one, set *output_file* and/or *output_dir*.

Executing a pipeline or pipeline step directly, or via run()

When calling a pipeline or step instance directly, or using the `run` method, you can specify individual parameter values manually. In this case, parameter files are not used. If you use `run` after instantiating with a parameter file (as is done when using the *call* method), the parameter file will be ignored.

```
# Instantiate the class. Do not provide a parameter file.
pipe = ExposurePipeline()

# Manually set any desired non-default parameter values
pipe.assign_wcs.skip = True
pipe.jump.rejection_threshold = 5
pipe.ramp_fit.override_gain = 'my_gain_file.asdf'
pipe.save_result = True
pipe.output_dir = '/my/data/pipeline_outputs'

# Run the pipeline
result = pipe('r0000101001001001001_01101_0001_WFI01_uncal.asdf')

# Or, execute the pipeline using the run method
result = pipe.run('r0000101001001001001_01101_0001_WFI01_uncal.asdf')
```

To run a single step:

```
from romancal.jump import JumpStep

# Instantiate the step
step = JumpStep()

# Set parameter values
step.rejection_threshold = 5
step.save_results = True
step.output_dir = '/my/data/jump_data'

# Execute by calling the instance directly
result = step('r0000101001001001001_01101_0001_WFI01_linearity.asdf')

# Or, execute using the run method
result = step.run('r0000101001001001001_01101_0001_WFI01_linearity.asdf')
```

Parameter Files

Parameter files can be used to specify parameter values when running a pipeline or individual steps. These values can be overridden either by the command line options and/or a local parameter file. See *Parameter Precedence* for a full description of how a parameter gets its final value.

A parameter file should be used when there are parameters a user wishes to change from the default version for a custom run of the step. To create a parameter file add `--save-parameters <filename.asdf>` to the command:

```
$ strun <step.class> <required-input-files> --save-parameters <filename.asdf>
```

For example, to save the parameters used for a run of the `ExposurePipeline` pipeline, use:

```
$ strun roman_elp r0000101001001001001_01101_0001_WFI01_uncal.asdf --save-parameters my_
↪exppars.asdf
```

Once saved, the file can be edited, removing parameters that should be left at their default values, and setting the remaining parameters to the desired values. Once modified, the new parameter file can be used:

```
$ strun my_exppars2.asdf r0000101001001001001_01101_0001_WFI01_uncal.asdf
```

Note that the parameter values will reflect whatever was set on the command-line, or through a specified local parameter file. In short, the values will be those actually used in the running of the step.

For more information about and editing of parameter files, see *ASDF Parameter Files*.

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

1.13.2 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.

Steps get their inputs from two sources:

- Configuration parameters come from the parameter file or the command line and are set as member variables on the Step object by the stpipe framework.
- Arguments are passed to the Step's `process` function as regular function arguments.

Parameters should be used to specify things that must be determined outside of the code by a user using the class. Arguments should be used to pass data that needs to go from one step to another as part of a larger pipeline. Another way to think about this is: if the user would want to examine or change the value, use a parameter.

The parameters are defined by the *Step.spec* member.

Input Files, Associations, and Directories

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 parameters `output_dir` and `output_file` which the user can use to specify the directory and file to save this model to. Since the `stpipe` architecture generally creates output file names, in general, it is expected that `output_file` be rarely specified, and that different sets of outputs be separated using `output_dir`.

Output Suffix

There are three ways a step's results can be written to a file:

1. Implicitly when a step is run from the command line or with `Step.from_cmdline`
2. Explicitly by specifying the parameter `save_results`
3. Explicitly by specifying a file name with the parameter `output_file`

In all cases, the file, or files, is/are created with an added suffix at the end of the base file name. By default this suffix is the class name of the step that produced the results. Use the `suffix` parameter to explicitly change the suffix.

The Python class

At a minimum, the Python Step class should inherit from `stpipe.Step`, implement a `process` method to do the actual work of the step and have a `spec` member to describe its parameters.

1. Objects from other Steps in a pipeline are passed as arguments to the `process` method.
2. The parameters described in *Configuring a Step* are available as member variables on `self`.
3. To support the caching suspend/resume feature of pipelines, images must be passed between steps as model objects. To ensure you're always getting a model object, call the model constructor on the parameter passed in. It is good idea to use a `with` statement here to ensure that if the input is a file path that the file will be appropriately closed.
4. Objects to pass to other Steps in the pipeline are simply returned from the function. To return multiple objects, return a tuple.
5. The parameters for the step are described in the `spec` member in the `configspec` format.
6. Declare any CRDS reference files used by the Step. (See *Interfacing with CRDS*).

```
from romancal.stpipe import RomanStep

from roman_datamodels.datamodels import ImageModel
from my_awesome_astronomy_library import combine

class ExampleStep(RomanStep):
    """
    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 parameters. It is in the `configspec` format defined in the `ConfigObj` library that stpipe uses.

The `configspec` format defines the types of the parameters, as well as allowing an optional tree structure.

The types of parameters are declared like this:

```
n_iterations = integer(1, 100)  # The number of iterations to run
factor = float()                # A multiplication factor
author = string()               # The author of the file
```

Note that each parameter may have a comment. This comment is extracted and displayed in help messages and docstrings etc.

Parameters can be grouped into categories using ini-file-like syntax:

```
[red]
offset = float()
scale = float()

[green]
offset = float()
scale = float()

[blue]
```

(continues on next page)

(continued from previous page)

```
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.

Configspec types

The following is a list of the commonly useful configspec types.

`integer`: matches integer values. Takes optional `min` and `max` arguments:

```
integer()
integer(3, 9)  # any value from 3 to 9
integer(min=0) # any positive value
integer(max=9)
```

`float`: matches float values Has the same parameters as the integer check.

`boolean`: matches boolean values: True or False.

`string`: matches any string. Takes optional keyword args `min` and `max` to specify min and max length of string.

`list`: matches any list. Takes optional keyword args `min`, and `max` to specify min and max sizes of the list. The list checks always return a list.

`force_list`: matches any list, but if a single value is passed in will coerce it into a list containing that value.

`int_list`: Matches a list of integers. Takes the same arguments as list.

`float_list`: Matches a list of floats. Takes the same arguments as list.

`bool_list`: Matches a list of boolean values. Takes the same arguments as list.

`string_list`: Matches a list of strings. Takes the same arguments as list.

`option`: matches any from a list of options. You specify this test with:

```
option('option 1', 'option 2', 'option 3')
```

Normally, steps will receive input files as parameters and return output files from their process methods. However, in cases where paths to files should be specified in the parameter file, there are some extra parameter types that stpipe provides that aren't part of the core configobj library.

`input_file`: Specifies an input file. Relative paths are resolved against the location of the parameter file. The file must also exist.

`output_file`: Specifies an output file. Identical to `input_file`, except the file doesn't have to already exist.

Interfacing with CRDS

If a Step uses CRDS to retrieve reference files, there are two things to do:

1. Within the `process` method, call `self.get_reference_file` or `self.get_reference_file_model` to get a reference file from CRDS. These methods take as input a) a model for the input file, whose metadata is used to do a CRDS bestref lookup, and b) a reference file type, which is just a string to identify the kind of reference file.
2. Declare the reference file types used by the Step in the `reference_file_types` member. This information is used by the stpipe framework for two purposes: a) to pre-cache the reference files needed by a Pipeline before any of the pipeline processing actually runs, and b) to add override parameters to the Step's configspec.

For each reference file type that the Step declares, an `override_*` parameter is added to the Step's configspec. For example, if a step declares the following:

```
reference_file_types = ['flat_field']
```

then the user can override the flat field reference file using the parameter file:

```
override_flat_field = /path/to/my_reference_file.asdf
```

or at the command line:

```
--override_flat_field=/path/to/my_reference_file.asdf
```

Pipelines

Writing a Pipeline

The basics of writing a Pipeline are just like *Writing a step*, but instead of inheriting from the `Step` class, one inherits from the `Pipeline` class.

In addition, a Pipeline subclass defines what its Steps are so that the framework can configure parameters for the individual Steps. This is done with the `step_defs` member, which is a dictionary mapping step names to step classes. This dictionary defines what the Steps are, but says nothing about their order or how data flows from one Step to the next. That is defined in Python code in the Pipeline's `process` method. By the time the Pipeline's `process` method is called, the Steps in `step_defs` will be instantiated as member variables.

For example, here is a pipeline with two steps: one that processes each chip of a multi-chip ASDF file, and another to combine the chips into a single image:

```
from romancal.stpipe import Pipeline

from roman_datamodels.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.
    """
```

(continues on next page)

(continued from previous page)

```
    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 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` 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` 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` as normal:

```
import logging

# ...
log = logging.getLogger()
```

(continues on next page)

(continued from previous page)

```
# If the log on its own won't emit, neither will it in the
# context of an stpipe step, so make sure the level is set to
# allow everything through
log.setLevel(logging.DEBUG)

def my_library_call():
    # ...
    log.info("I want to make note of something")
    # ...
```

romancal.stpipe Package

Classes

RomanPipeline(*args, **kwargs)	See `Step.__init__` for the parameters.
RomanStep([name, parent, config_file, ...])	Base class for Roman calibration pipeline steps.

RomanPipeline

class romancal.stpipe.**RomanPipeline**(**args*, ***kwargs*)

Bases: `Pipeline`, *RomanStep*

See `Step.__init__` for the parameters.

RomanStep

class romancal.stpipe.**RomanStep**(*name=None*, *parent=None*, *config_file=None*, *_validate_kwds=True*, ***kws*)

Bases: `Step`

Base class for Roman calibration pipeline steps.

Create a `Step` instance.

Parameters

- **name** (*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 or pathlib.Path, optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- ****kws** (*dict*) – Additional parameters to set. These will be set as member variables on the new Step instance.

Attributes Summary

spec	

Methods Summary

finalize_result(model, reference_files_used)	Hook that allows the Step to set metadata on the output model before save.
record_step_status(model, step_name[, success])	Record step completion status in the model's metadata.
remove_suffix(name)	Remove any Roman step-specific suffix from the given filename.

Attributes Documentation

spec

```
output_ext =  string(default='.asdf')    # Default type of output
```

Methods Documentation

finalize_result(*model*, *reference_files_used*)

Hook that allows the Step to set metadata on the output model before save.

Parameters

- **model** (*roman_datamodels.datamodels.DataModel*) – Output model.
- **reference_files_used** (*list of tuple(str, str)*) – List of reference files used. The first element of each tuple is the reftype code, the second element is the filename.

record_step_status(*model*, *step_name*, *success=True*)

Record step completion status in the model's metadata.

Parameters

- **model** (*roman_datamodels.datamodels.DataModel*) – Output model.
- **step_name** (*str*) – Calibration step name.
- **success** (*bool*) – If True, then the step was run successfully.

remove_suffix(*name*)

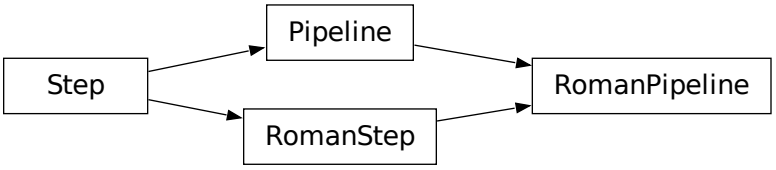
Remove any Roman step-specific suffix from the given filename.

Parameters
name (*str*) – Filename.

Returns
Filename with step suffix removed.

Return type
str

Class Inheritance Diagram



1.14 About datamodels

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 ASDF files or some future file format.

The detailed datamodel structure and specifics are contained in the documentation included with the `roman_datamodels` package found here.

Each model instance is created to contain a variety of attributes and data that are needed for analysis or to propagate information about the file and the contents of the file. 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

Along with data arrays the datamodel also contains information about the observation that can include the observation program, exposure information, pointing information and processing steps.

1.15 Working with models

1.15.1 Reading a data model

If you have an existing data file it is straightforward to access the file using python.

```
>>> from roman_datamodels import datamodels as rdm
>>> fn = 'r0019106003005004023_03203_0034_WFI01_cal.asdf'
>>> data_file = rdm.open(fn)
>>> type(data_file)
roman_datamodels.datamodels.ImageModel
```

Where the output of the type command tells you that you have imported an ImageModel from roman_datamodels,

1.15.2 Creating a data model from scratch

To create a new `ImageModel`, you can just

```
>>> from roman_datamodels import datamodels as rdm
>>> from roman_datamodels.maker_utils import mk_datamodel

>>> image_model = mk_datamodel(rdm.ImageModel)
>>> type(image_model)
<class 'roman_datamodels.datamodels._datamodels.ImageModel'>
```

Warning: The values in the file generated by create_wfi_image are intended to be clearly incorrect and should be replaced if the file is intended to be used for anything besides a demonstration.

1.15.3 Saving a data model to a file

Simply call the `save` method on the model instance:

```
>>> image_model.save("myimage.asdf")
PosixPath('myimage.asdf')
```

Note: This `save` always clobbers the output file. For now the only format supported is ASDF.

1.15.4 Creating a data model from a file

The `roman_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 an ASDF file
- 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.

```
>>> with rdm.open("myimage.asdf") as im:
...     assert isinstance(im, rdm.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

```
>>> with rdm.ImageModel("myimage.asdf") as im:
...     pass  # raises exception if myimage.asdf is not an image file
```

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

1.15.5 Copying a model

To create a new model based on another model, simply use its `copy` method. This will perform a deep-copy: that is, no changes to the original model will propagate to the new model

```
>>> new_model = image_model.copy()
```

1.15.6 Looking at the contents of a model

You can examine the contents of your model from within python using

```
>>> print("\n".join("{: >20}\t{}".format(k, v) for k, v in image_model.items()), "\n")
meta.calibration_software_version   9.9.0
meta.sdf_software_version   7.7.7
       meta.filename        dummy value
      meta.file_date        2020-01-01T00:00:00.000
     meta.model_type        ImageModel
         meta.origin        STSCI
meta.prd_software_version   8.8.8
      meta.telescope        ROMAN
  meta.aperture.name        WFI_06_FULL
meta.aperture.position_angle        30.0
...
```

or you can print specifics

```
>>> print("\n".join("{: >20}\t{}".format(k, v) for k, v in image_model.meta.wcsinfo.
↪items()), "\n")
              v2_ref        -999999
              v3_ref        -999999
             vparity        -999999
            v3yangle        -999999
              ra_ref        -999999
             dec_ref        -999999
            roll_ref        -999999
            s_region        dummy value
```

Note: These will be incorporated as methods in the data models in a future release.

1.16 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.start_time)
```

Metadata values are automatically type-checked against the schema when they are set. Therefore, setting a attribute which expects a number to a string will raise an exception.

```
>>> from roman_datamodels import datamodels as rdm
>>> from roman_datamodels.maker_utils import mk_datamodel

>>> model = mk_datamodel(rdm.ImageModel)
>>> model.meta.target.ra = "foo"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/ddavis/miniconda3/envs/rcal_dev/lib/python3.9/site-packages/roman_
↪datamodels/stnode.py", line 183, in __setattr__
    if schema is None or _validate(key, value, schema, self.ctx):
  File "/Users/ddavis/miniconda3/envs/rcal_dev/lib/python3.9/site-packages/roman_
↪datamodels/stnode.py", line 97, in _validate
    return _value_change(attr, tagged_tree, schema, False, strict_validation, ctx)
  File "/Users/ddavis/miniconda3/envs/rcal_dev/lib/python3.9/site-packages/roman_
↪datamodels/stnode.py", line 68, in _value_change
    raise jsonschema.ValidationError(errmsg)
jsonschema.exceptions.ValidationError: While validating ra the following error occurred:
'foo' is not of type 'number'

Failed validating 'type' in schema:
    {'$schema': 'http://stsci.edu/schemas/asdf-schema/0.1.0/asdf-schema',
     'archive_catalog': {'datatype': 'float',
                         'destination': ['ScienceCommon.ra']},
     'sdf': {'source': {'origin': 'PSS:fixed_target.ra_computed'},
             'special_processing': 'VALUE_REQUIRED'},
     'title': 'Target RA at mid time of exposure',
     'type': 'number'}

On instance:
    'foo'
```

The set of available metadata elements is defined in a YAML Schema that is installed with roman_datamodels from the RAD (Roman Attribute Dictionary).

There is also a utility method for finding the schema associated with a given model.

```
>>> raw_science = mk_datamodel(rdm.ScienceRawModel)  # Create a model of the desired type
>>> print(raw_science.schema_uri)  # find the associated Schema
asdf://stsci.edu/datamodels/roman/schemas/wfi_science_raw-1.0.0
```

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(raw_science.meta['visit']['start_time'])
2020-01-01T00:00:00.000
>>> print(raw_science.meta.visit.start_time)
2020-01-01T00:00:00.000
```

In addition the times are stored as Astropy time objects and so the date can be displayed using various formats

```
>>> print(raw_science.meta.visit.start_time.iso)
2020-01-01 00:00:00.000
>>> print(raw_science.meta.visit.start_time.mjd)
```

(continues on next page)

(continued from previous page)

```
58849.0
>>> print(raw_science.meta.visit.start_time.yday)
2020:001:00:00:00.000
```

1.17 Working with Roman datamodels and ASDF files

Please refer to Roman Documentation for more details about `roman_datamodels`.

This section assumes that you are familiar with the ASDF standard format. If that's not the case, a good starting point would be to go over the ASDF Documentation.

If you have installed the roman calibration pipeline you should also have access to the standalone tool asdfinfo which allows access to ASDF (and roman) files from the terminal prompt,:

```
asdftool info r0000101001001001001_01101_0001_WFI16_cal.asdf
root (AsdfObject)
├─asdf_library (Software)
│ ├─author (str): The ASDF Developers
...
```

also useful is:

```
asdftool help
usage: asdftool [-h] [--verbose] {help,explode,implode,extract,defragment,diff,edit,
↪remove-hdu,info,extensions,tags,to_yaml} ...

Commandline utilities for managing ASDF files.

optional arguments:
  -h, --help            show this help message and exit
  --verbose, -v         Increase verbosity

subcommands:

  {help,explode,implode,extract,defragment,diff,edit,remove-hdu,info,extensions,tags,to_
↪yaml}
    help                Display usage information
    explode             Explode a ASDF file.
    implode             Implode a ASDF file.
    extract             Extract ASDF extensions in ASDF-in-FITS files into pure ASDF␣
↪files
    defragment          Defragment an ASDF file..
    diff                Report differences between two ASDF files
    remove-hdu          Remove ASDF extension from ASDF-in-FITS file
    info                Print a rendering of an ASDF tree.
    extensions          Show information about installed extensions
    tags                List currently available tags
    to_yaml             Convert as ASDF file to pure YAML.
```

which gives a list of possible actions one of the more useful can be:

```
asdftool edit file.asdf
```

Which will open the file in an editor you have set via the EDITOR environment variable. A more complete description of the options can be found in the asdftool documentation.

To access the files via a python session,

```
import roman_datamodels as rdm
import asdf
with rdm.open('r0000101001001001001_01101_0001_WFI16_cal.asdf') as model:
    <Manipulate the files>

with asdf.open('r0000101001001001001_01101_0001_WFI16_cal.asdf', copy_arrays=True) as␣
↪model:
    <Manipulate the files>
```

Once the files are loaded you can access various attributes. Below is a table showing how to access various properties using the roman_datamodels and the asdf.open methods,

Roman Datamodels	ASDF

`rdm_a.meta`
`rdm_a.meta.aperture`
`rdm_a.meta.aperture.position_angle`
`120`
	`asdf_a.tree['roman']['meta']`
	`asdf_a.tree['roman']['meta']['aperture']`
	`asdf_a.tree['roman']['meta']['aperture']['position_angle']`
	`120`

You can also update or modify the metadata in Roman datamodels

```
rdm_a.meta.aperture.position_angle = 120.21
rdm_a.meta.aperture.position_angle
120.21
```

The ASDF equivalent is

```
asdf_a.tree['roman']['meta']['aperture']['position_angle'] = 120.05
asdf_a.tree['roman']['meta']['aperture']['position_angle']
120.05
```

Hint: If you trigger an error, "ValueError: assignment destination is read-only" make sure the asdf file was opened with copy_arrays=True, or with mode='rw'

You can also access and modify the data arrays

Listing 1: Roman Datamodels

```
rdm_a.data
<array (unloaded) shape: [4096, 4096] dtype: float32>

rdm_a.data[10,11]
0.0
```

(continues on next page)

(continued from previous page)

```
rdm_a.data[10,11] = 122.1
rdm_a.data[10,11]
122.1
```

or by modifying the ASDF tree,

Listing 2: ASDF

```
asdf_a.tree['roman']['data']
<array (unloaded) shape: [4096, 4096] dtype: float32>

asdf_a.tree['roman']['data'][10,11]
0.0

asdf_a.tree['roman']['data'][10,11] = 3.14159
asdf_a.tree['roman']['data'][10,11]
3.14159
```

1.17.1 Using the info method

You can examine a roman data model using the info and search methods provided from the asdf package. The info fuction will print a representation of the asdf tree.

```
>>> from roman_datamodels import datamodels as rdm
>>> d_uncal = rdm.open('r0000101001001001001_01101_0001_WFI01_uncal.asdf')
>>> d_uncal.info()
root (AsdfObject)
├─asdf_library (Software)
│  ├─author (str): The ASDF Developers
│  ├─homepage (str): http://github.com/asdf-format/asdf
│  ├─name (str): asdf
│  └─version (str): 2.8.1
├─history (dict)
│  └─extensions (list)
│    ├─[0] (ExtensionMetadata) ...
│    ├─[1] (ExtensionMetadata) ...
│    └─[2] (ExtensionMetadata) ...
└─roman (WfiScienceRaw)
  ├─meta (dict)
  │  ├─aperture (Aperture) ...
  │  ├─cal_step (L2CalStep) ...
  │  ├─calibration_software_version (str): 0.4.3.dev89+gca5771d
  │  ├─coordinates (Coordinates) ...
  │  ├─crds_context_used (str): roman_0020.pmap
  │  ├─crds_software_version (str): 11.5.0
  │  ├─ephemeris (Ephemeris) ...
  │  ├─exposure (Exposure) ...
  │  └─17 not shown
  └─data (NDArrayType): shape=(8, 4096, 4096), dtype=uint16
Some nodes not shown.
```

The info command also gives you control over the number of lines displayed by passing the argument `max_rows`. As an integer, `max_rows` will be interpreted as an overall limit on the number of displayed lines. If `max_rows` is a tuple, then each member limits lines per node at the depth corresponding to its tuple index. For example, to show all top-level nodes and 5 of each's children:

```
>>> d_uncal.info(max_rows=(None,5))
root (AsdfObject)
├─asdf_library (Software)
│  ├─author (str): The ASDF Developers
│  ├─homepage (str): http://github.com/asdf-format/asdf
│  ├─name (str): asdf
│  └─version (str): 2.8.1
├─history (dict)
│  └─extensions (list) ...
└─roman (WfiScienceRaw)
   ├─meta (dict) ...
   └─data (NDArrayType): shape=(8, 4096, 4096), dtype=uint16
Some nodes not shown.
```

Or you can use the asdf.info method to view the contents of the tree

```
>> asdf.info(d_uncal)
```

Will print the same information as the above `d_uncal.info` command but also gives you enhanced capabilities. For instance you can display the first three lines for each of the meta entries,

```
>>> asdf.info(d_uncal.meta, max_rows=(None, 3))
root (DNode)
├─aperture (Aperture)
│  ├─name (str): WFI_CEN
│  └─position_angle (int): 120
├─cal_step (L2CalStep)
│  ├─assign_wcs (str): INCOMPLETE
│  ├─flat_field (str): INCOMPLETE
│  └─6 not shown
├─calibration_software_version (str): 0.4.3.dev89+gca5771d
├─coordinates (Coordinates)
│  └─reference_frame (str): ICRS
├─crds_context_used (str): roman_0020.pmap
├─crds_software_version (str): 11.5.0
├─ephemeris (Ephemeris)
│  ├─earth_angle (float): 3.3161255787892263
│  ├─moon_angle (float): 3.3196162372932148
│  └─10 not shown
...
```

or you can concentrate on a given attribute. To list all the attributes in `cal_step` without listing the values,

```
>>> asdf.info(d_uncal.meta.cal_step,max_rows=(None, 3),show_values=False)
root (L2CalStep)
├─assign_wcs (str)
├─flat_field (str)
├─dark (str)
├─dq_init (str)
```

(continues on next page)

(continued from previous page)

```
├─jump (str)
├─linearity (str)
├─ramp_fit (str)
└─saturation (str)
```

More information on the info method can be found in the ASDF documentation at rendering the ASDF trees.

1.17.2 Using the search method

You can also use the search method to find attributes,

```
>>> d_uncal.search('cal_step')
root (AsdfObject)
└─roman (WfiScienceRaw)
  └─meta (dict)
    └─cal_step (L2CalStep)
```

or a a general search for all attributes with cal in the name

```
>>> d_uncal.search('cal')
root (AsdfObject)
└─roman (WfiScienceRaw)
 └─meta (dict)
   ├─cal_step (L2CalStep)
   ├─calibration_software_version (str): 0.4.3.dev89+gca5771d
   ├─instrument (WfiMode)
   │ └─optical_element (str): F158
   └─velocity_aberration (VelocityAberration)
     └─scale_factor (float): 0.9999723133902021
```

This will do a regular expression search for `cal` in the attribute name. More information on using regular expressions in the search method can be found in the ASDF documentation linked below.

To search only within the meta tree,

```
>>> d_uncal.search('cal_')['roman']['meta']
meta (dict)
├─cal_step (L2CalStep)
└─instrument (WfiMode)
  └─optical_element (str): F158
```

You can also use the search method to find attributes by type in the asdf tree. For instance, you can find all integers, floats, or booleans by using the type keyword,

```
>>> d_uncal.search(type=bool)
root (AsdfObject)
└─roman (WfiScienceRaw)
  └─meta (dict)
    ├─exposure (Exposure)
    │ └─data_problem (bool): False
    └─visit (Visit)
      ├─internal_target (bool): False
```

(continues on next page)

(continued from previous page)

```
      └─target_of_opportunity (bool): False

>>> d_uncal.search(type=bool, value=True)
No results found.
```

More information and options for the search method can be found in the ASDF documentation here.

1.18 ModelContainer

class romancal.datamodels.container.**ModelContainer**(*init=None*, *asn_exptypes=None*, *asn_n_members=None*, *iscopy=False*, *memmap=False*, *return_open=True*, *save_open=True*)

A container for holding DataModels.

This functions like a list for holding DataModel objects. It can be iterated through like a list and the datamodels within the container can be addressed by index. Additionally, the datamodels can be grouped by exposure.

Parameters

- **init** (path to ASN file, list of either datamodels or path to ASDF files, or `None`) – If `None`, then an empty *ModelContainer* instance is initialized, to which datamodels can later be added via the `insert()`, `append()`, or `extend()` method.
- **iscopy** (*bool*) – Presume this model is a copy. Members will not be closed when the model is closed/garbage-collected.
- **memmap** (*bool*) – Open ASDF file binary data using memmap (default: False)
- **return_open** (*bool*) – (optional) See notes below on usage.
- **save_open** (*bool*) – (optional) See notes below on usage.

Examples

To load a list of ASDF files into a *ModelContainer*:

```
container = ModelContainer(
    [
        "/path/to/file1.asdf",
        "/path/to/file2.asdf",
        ...,
        "/path/to/fileN.asdf"
    ]
)
```

To load a list of open Roman DataModels into a *ModelContainer*:

```
import roman_datamodels.datamodels as rdm
data_list = [
        "/path/to/file1.asdf",
        "/path/to/file2.asdf",
        ...,
```

(continues on next page)

(continued from previous page)

```
        "/path/to/fileN.asdf"
    ]
datamodels_list = [rdm.open(x) for x in data_list]
container = ModelContainer(datamodels_list)
```

To load an ASN file into a *ModelContainer*:

```
asn_file = "/path/to/asn_file.json"
container = ModelContainer(asn_file)
```

In any of the cases above, the content of each file in a *ModelContainer* can be accessed by iterating over its elements. For example, to print out the filename of each file, we can run:

```
for model in container:
    print(model.meta.filename)
```

Additionally, *ModelContainer* can be used with context manager:

```
with ModelContainer(asn_file) as asn:
    # do stuff
```

Notes

The optional parameters `save_open` and `return_open` can be provided to control how the `DataModel` are used by the *ModelContainer*. If `save_open` is set to `False`, each input `DataModel` instance in `init` will be written out to disk and closed, then only the filename for the `DataModel` will be used to initialize the *ModelContainer* object. Subsequent access of each member will then open the `DataModel` file to work with it. If `return_open` is also `False`, then the `DataModel` will be closed when access to the `DataModel` is completed. The use of these parameters can minimize the amount of memory used by this object during processing.

> **Warning:** Input files will be updated in-place with new `meta` attribute values when ASN table's members contain additional attributes.

append(*model*)

close()

Close all datamodels.

copy(*memo=None*)

Returns a deep copy of the models in this model container.

property crds_observatory

Get the CRDS observatory for this container. Used when selecting step/pipeline parameter files when the container is a pipeline input.

Return type
str

extend(*input_object*)

from_asn(*asn_data*, *asn_file_path=None*)

Load ASDF files from a Roman association file.

Parameters

- **asn_data** (Association) – Association dictionary.
- **asn_file_path** (*str*) – Filepath of the association, if known.

get_crds_parameters()

Get parameters used by CRDS to select references for this model.

Return type
dict

get_sections()

Iterator to return the sections from all members of the container.

insert(*index*, *model*)

merge_tree(*a*, *b*)

Merge elements from tree b into tree a.

property models_grouped

Returns a list of a list of datamodels grouped by exposure. Assign an ID grouping 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 meta.observation.observation meta.observation.visit meta.observation.visit_file_group meta.observation.visit_file_sequence meta.observation.visit_file_activity meta.observation.exposure

pop(*index=-1*)

static read_asn(*filepath*)

Load ASDF files from a Roman association file.

Parameters
filepath (*str*) – The path to an association file.

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

Write out models in container to ASDF.

Parameters

- **path** (*str or func or 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*) – 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*) – Alternate function to save each model instead of the models *save* method. Takes one argument, the model, and keyword argument idx for an index.

Note: Additional parameters provided via `**kwargs` are passed on to `roman_datamodels.datamodels.DataModel.to_asdf`

Returns
output_paths – List of output file paths of where the models were saved.

Return type
[str[, ...]]

`set_buffer`(*buffer_size*, *overlap=None*)

Set buffer size for scrolling section-by-section access.

Parameters

- **buffer_size** (*float, None*) – Define size of buffer in MB for each section. If `None`, a default buffer size of 1MB will be used.
- **overlap** (*int, optional*) – Define the number of rows of overlaps between sections. If `None`, no overlap will be used.

1.19 Generating Static Previews

Roman archiving requires static preview images for viewing and selecting images, with the following requirements for each `ImageModel`:

- 1080p x 1080p preview image
- 300p x 300p thumbnail image
- output as PNG files
- 90th percentile linear histogram stretch
- using `afmhot` colormap
- overlay indicating orientation

The `roman_static_preview` script creates downsampled images from ASDF files containing an `ImageModel`, with an optional compass rose overlayed onto the image indicating orientation.

1.19.1 Installation

The requirements for this script are not installed by default as part of `romancal`; install with the `sdp` extra to include them.

```
pip install "romancal[sdp]"
```

1.19.2 Usage

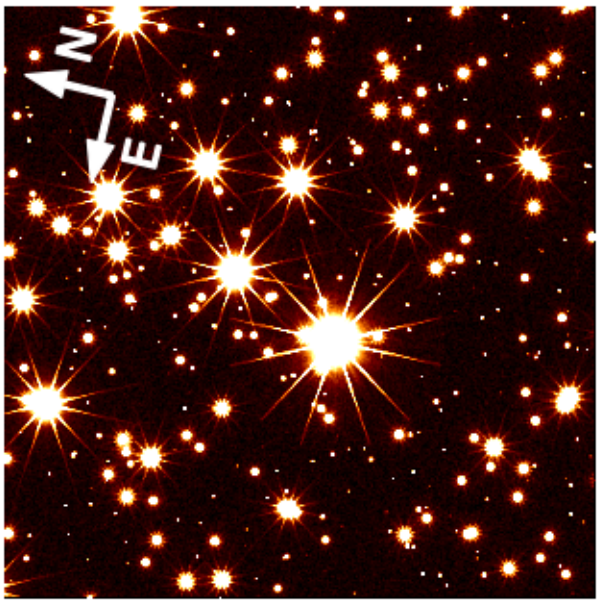
`roman_static_preview` includes two convenience commands, `preview` and `thumbnail`, that set default options to the static preview requirements.

```
 roman_static_preview preview --help
Usage: roman_static_preview preview [OPTIONS] INPUT [OUTPUT] [SHAPE]...

  create a preview image with a north arrow overlay indicating orientation

Arguments:
  INPUT       path to ASDF file with 2D image data  [required]
  [OUTPUT]    path to output image file
  [SHAPE]...  desired pixel resolution of output image  [default: 1080, 1080]

Options:
  --compass / --no-compass  whether to draw a north arrow on the image
                            [default: compass]
  --help                    Show this message and exit.
```

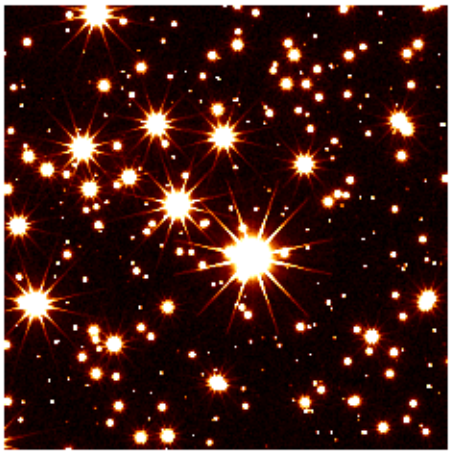
```
 roman_static_preview thumbnail --help
Usage: roman_static_preview thumbnail [OPTIONS] INPUT [OUTPUT] [SHAPE]...

Arguments:
  INPUT       path to ASDF file with 2D image data  [required]
  [OUTPUT]    path to output image file
  [SHAPE]...  desired pixel resolution of output image  [default: 300, 300]

Options:
  --compass / --no-compass  whether to draw a north arrow on the image
                            [default: no-compass]
  --help                    Show this message and exit.
```

1.19.3 Examples

```
roman_static_preview preview r0000501001001001001_01101_0001_WFI01_cal.asdf␣
↪r0000501001001001001_01101_0001_WFI01_cal.png 400 400
```




```
roman_static_preview thumbnail r0000501001001001001_01101_0001_WFI01_cal.asdf␣
↪r0000501001001001001_01101_0001_WFI01_cal_thumb.png
```



1.19.4 using `stpreview` directly

The `roman_static_preview` script is merely a wrapper over `stpreview to`, which offers more options for fine-grained control of the output image. `stpreview` offers the `to` and `by` commands (for resampling `to` a desired image shape, or `by` a desired factor, respectively). Refer to the documentation for usage instructions.

PYTHON MODULE INDEX

r

romancal.assign_wcs, 44
romancal.dark_current, 24
romancal.datamodels.container, 177
romancal.dq_init, 12
romancal.flatfield, 48
romancal.flux, 55
romancal.flux.flux_step, 54
romancal.jump, 29
romancal.lib.psf, 62
romancal.linearity, 21
romancal.outlier_detection, 106
romancal.outlier_detection.outlier_detection, 101
romancal.outlier_detection.outlier_detection_step, 97
romancal.photom, 51
romancal.pipeline, 137
romancal.ramp_fitting, 39
romancal.resample, 121
romancal.resample.resample, 114
romancal.resample.resample_step, 112
romancal.resample.resample_utils, 118
romancal.saturation, 16
romancal.skymatch, 91
romancal.skymatch.region, 90
romancal.skymatch.skyimage, 84
romancal.skymatch.skymatch, 81
romancal.skymatch.skymatch_step, 80
romancal.skymatch.skystatistics, 89
romancal.source_detection, 60
romancal.stpipe, 166
romancal.tweakreg, 74
romancal.tweakreg.astrometric_utils, 72
romancal.tweakreg.tweakreg_step, 71

INDEX

A

abs_deriv() (*in module romancal.outlier_detection.outlier_detection*), 102
append() (*romancal.datamodels.container.ModelContainer method*), 178
append() (*romancal.skymatch.skyimage.SkyGroup method*), 85
AssignWcsStep (*class in romancal.assign_wcs*), 44

B

blot_median() (*romancal.outlier_detection.outlier_detection.OutlierDetection method*), 104
build_driz_weight() (*in module romancal.resample.resample_utils*), 118
build_mask() (*in module romancal.resample.resample_utils*), 118

C

calc_bounding_polygon() (*romancal.skymatch.skyimage.SkyImage method*), 87
calc_gwcs_pixmap() (*in module romancal.resample.resample_utils*), 118
calc_sky() (*romancal.skymatch.skyimage.SkyGroup method*), 85
calc_sky() (*romancal.skymatch.skyimage.SkyImage method*), 87
calc_sky() (*romancal.skymatch.skystatistics.SkyStats method*), 89
class_alias (*romancal.outlier_detection.outlier_detection_step.OutlierDetectionStep attribute*), 98
class_alias (*romancal.outlier_detection.OutlierDetectionStep attribute*), 107
class_alias (*romancal.pipeline.ExposurePipeline attribute*), 138
class_alias (*romancal.pipeline.HighLevelPipeline attribute*), 139
class_alias (*romancal.resample.resample_step.ResampleStep attribute*), 113
class_alias (*romancal.resample.ResampleStep attribute*), 122
class_alias (*romancal.skymatch.skymatch_step.SkyMatchStep attribute*), 81
class_alias (*romancal.skymatch.SkyMatchStep attribute*), 93
class_alias (*romancal.tweakreg.tweakreg_step.TweakRegStep attribute*), 71
class_alias (*romancal.tweakreg.TweakRegStep attribute*), 75
close() (*romancal.datamodels.container.ModelContainer method*), 178
compute_AET_entry() (*romancal.skymatch.region.Edge method*), 90
compute_GET_entry() (*romancal.skymatch.region.Edge method*), 90
compute_radius() (*in module romancal.tweakreg.astrometric_utils*), 72
copy() (*romancal.datamodels.container.ModelContainer method*), 178
copy() (*romancal.skymatch.skyimage.SkyImage method*), 87
crds_observatory (*romancal.datamodels.container.ModelContainer property*), 178
create_astrometric_catalog() (*in module romancal.tweakreg.astrometric_utils*), 72
create_fully_saturated_zeroed_image() (*romancal.pipeline.ExposurePipeline method*), 138
create_gridded_psf_model() (*in module romancal.lib.psf*), 62
create_median() (*romancal.outlier_detection.outlier_detection.OutlierDetection method*), 104

D

DarkCurrentStep (*class in romancal.dark_current*), 25
DataAccessor (*class in romancal.skymatch.skyimage*), 84
decode_context() (*in module romancal.resample.resample_utils*), 119
default_suffix (*romancal.outlier_detection.outlier_detection.OutlierDetection attribute*), 103

detect_outliers() (*romancal.outlier_detection.outlier_detection.OutlierDetection method*), 104
do_detection() (*romancal.outlier_detection.outlier_detection.OutlierDetection method*), 104
do_drizzle() (*romancal.resample.resample.ResampleData method*), 116
dq_to_boolean_mask() (*in module romancal.lib.psf*), 65
DQInitStep (*class in romancal.dq_init*), 12
drizzle_arrays() (*romancal.resample.resample.ResampleData static method*), 116

E

Edge (*class in romancal.skymatch.region*), 90
ExposurePipeline (*class in romancal.pipeline*), 137
extend() (*romancal.datamodels.container.ModelContainer method*), 178

F

finalize_result() (*romancal.stpipe.RomanStep method*), 167
fit_psf_to_image_model() (*in module romancal.lib.psf*), 63
flag_cr() (*in module romancal.outlier_detection.outlier_detection*), 102
FlatFieldStep (*class in romancal.flatfield*), 48
FluxStep (*class in romancal.flux*), 56
FluxStep (*class in romancal.flux.flux_step*), 54
from_asn() (*romancal.datamodels.container.ModelContainer method*), 178

G

get_catalog() (*in module romancal.tweakreg.astrometric_utils*), 73
get_crds_parameters() (*romancal.datamodels.container.ModelContainer method*), 179
get_data() (*romancal.skymatch.skyimage.DataAccessor method*), 84
get_data() (*romancal.skymatch.skyimage.NDArrayInMemoryAccessor method*), 84
get_data() (*romancal.skymatch.skyimage.NDArrayMappedAccessor method*), 84
get_data_shape() (*romancal.skymatch.skyimage.DataAccessor method*), 84
get_data_shape() (*romancal.skymatch.skyimage.NDArrayInMemoryAccessor method*), 84
get_data_shape() (*romancal.skymatch.skyimage.NDArrayMappedAccessor method*), 84
get_edges() (*romancal.skymatch.region.Polygon method*), 90
get_sections() (*romancal.datamodels.container.ModelContainer method*), 179

H

HighLevelPipeline (*class in romancal.pipeline*), 138

I

id (*romancal.skymatch.skyimage.SkyGroup property*), 85
id (*romancal.skymatch.skyimage.SkyImage property*), 87
image (*romancal.skymatch.skyimage.SkyImage property*), 88
image_shape (*romancal.skymatch.skyimage.SkyImage property*), 88
insert() (*romancal.datamodels.container.ModelContainer method*), 179
insert() (*romancal.skymatch.skyimage.SkyGroup method*), 85
intersection() (*romancal.skymatch.region.Edge method*), 90
intersection() (*romancal.skymatch.skyimage.SkyGroup method*), 85
intersection() (*romancal.skymatch.skyimage.SkyImage method*), 88
is_parallel() (*romancal.skymatch.region.Edge method*), 90
is_sky_valid (*romancal.skymatch.skyimage.SkyImage property*), 88

J

JumpStep (*class in romancal.jump*), 29

L

LinearityStep (*class in romancal.linearity*), 21

M

make_output_wcs() (*in module romancal.resample.resample_utils*), 120
mask (*romancal.skymatch.skyimage.SkyImage property*), 88
match() (*in module romancal.skymatch.skymatch*), 81
merge_tree() (*romancal.datamodels.container.ModelContainer method*), 179
ModelContainer (*class in romancal.datamodels.container*), 177

models_grouped (*romancal.datamodels.container.ModelContainer property*), 179
module
    romancal.assign_wcs, 44
    romancal.dark_current, 24
    romancal.datamodels.container, 177
    romancal.dq_init, 12
    romancal.flatfield, 48
    romancal.flux, 55
    romancal.flux.flux_step, 54
    romancal.jump, 29
    romancal.lib.psf, 62
    romancal.linearity, 21
    romancal.outlier_detection, 106
    romancal.outlier_detection.outlier_detection, 101
    romancal.outlier_detection.outlier_detection_step, 97
    romancal.photom, 51
    romancal.pipeline, 137
    romancal.ramp_fitting, 39
    romancal.resample, 121
    romancal.resample.resample, 114
    romancal.resample.resample_step, 112
    romancal.resample.resample_utils, 118
    romancal.saturation, 16
    romancal.skymatch, 91
    romancal.skymatch.region, 90
    romancal.skymatch.skyimage, 84
    romancal.skymatch.skymatch, 81
    romancal.skymatch.skymatch_step, 80
    romancal.skymatch.skystatistics, 89
    romancal.source_detection, 60
    romancal.stpipe, 166
    romancal.tweakreg, 74
    romancal.tweakreg.astrometric_utils, 72
    romancal.tweakreg.tweakreg_step, 71

N

NDArrayInMemoryAccessor (*class in romancal.skymatch.skyimage*), 84
NDArrayMappedAccessor (*class in romancal.skymatch.skyimage*), 84
next (*romancal.skymatch.region.Edge property*), 90

O

ols() (*romancal.ramp_fitting.RampFitStep method*), 41
ols_cas22() (*romancal.ramp_fitting.RampFitStep method*), 41
OutlierDetection (*class in romancal.outlier_detection.outlier_detection*), 102
OutlierDetectionStep (*class in romancal.outlier_detection*), 106
OutlierDetectionStep (*class in romancal.outlier_detection.outlier_detection_step*), 97
OutputTooLargeError, 114

P

PhotomStep (*class in romancal.photom*), 51
pix_area (*romancal.skymatch.skyimage.SkyImage property*), 88
poly_area (*romancal.skymatch.skyimage.SkyImage property*), 88
Polygon (*class in romancal.skymatch.region*), 90
polygon (*romancal.skymatch.skyimage.SkyGroup property*), 86
polygon (*romancal.skymatch.skyimage.SkyImage property*), 88
pop() (*romancal.datamodels.container.ModelContainer method*), 179
process() (*romancal.assign_wcs.AssignWcsStep method*), 45
process() (*romancal.dark_current.DarkCurrentStep method*), 26
process() (*romancal.dq_init.DQInitStep method*), 13
process() (*romancal.flatfield.FlatFieldStep method*), 49
process() (*romancal.flux.flux_step.FluxStep method*), 55
process() (*romancal.flux.FluxStep method*), 57
process() (*romancal.jump.JumpStep method*), 30
process() (*romancal.linearity.LinearityStep method*), 22
process() (*romancal.outlier_detection.outlier_detection_step.OutlierDete method*), 99
process() (*romancal.outlier_detection.OutlierDetectionStep method*), 108
process() (*romancal.photom.PhotomStep method*), 52
process() (*romancal.pipeline.ExposurePipeline method*), 138
process() (*romancal.pipeline.HighLevelPipeline method*), 139
process() (*romancal.ramp_fitting.RampFitStep method*), 41
process() (*romancal.resample.resample_step.ResampleStep method*), 114
process() (*romancal.resample.ResampleStep method*), 123
process() (*romancal.saturation.SaturationStep method*), 17
process() (*romancal.skymatch.skymatch_step.SkyMatchStep method*), 81
process() (*romancal.skymatch.SkyMatchStep method*), 93
process() (*romancal.source_detection.SourceDetectionStep method*), 62

process() (*romancal.tweakreg.tweakreg_step.TweakRegStep method*), 71
process() (*romancal.tweakreg.TweakRegStep method*), 76

R

radec (*romancal.skymatch.skyimage.SkyGroup property*), 86
radec (*romancal.skymatch.skyimage.SkyImage property*), 88
RampFitStep (*class in romancal.ramp_fitting*), 40
read_asn() (*romancal.datamodels.container.ModelContainer static method*), 179
record_step_status() (*romancal.stpipe.RomanStep method*), 167
refcat (*romancal.tweakreg.tweakreg_step.TweakRegStep attribute*), 71
refcat (*romancal.tweakreg.TweakRegStep attribute*), 75
reference_file_types (*romancal.assign_wcs.AssignWcsStep attribute*), 44
reference_file_types (*romancal.dark_current.DarkCurrentStep attribute*), 25
reference_file_types (*romancal.dq_init.DQInitStep attribute*), 13
reference_file_types (*romancal.flatfield.FlatFieldStep attribute*), 49
reference_file_types (*romancal.flux.flux_step.FluxStep attribute*), 55
reference_file_types (*romancal.flux.FluxStep attribute*), 57
reference_file_types (*romancal.jump.JumpStep attribute*), 30
reference_file_types (*romancal.linearity.LinearityStep attribute*), 22
reference_file_types (*romancal.photom.PhotomStep attribute*), 52
reference_file_types (*romancal.ramp_fitting.RampFitStep attribute*), 40
reference_file_types (*romancal.resample.resample_step.ResampleStep attribute*), 113
reference_file_types (*romancal.resample.ResampleStep attribute*), 122
reference_file_types (*romancal.saturation.SaturationStep attribute*), 16
reference_file_types (*romancal.skymatch.skymatch_step.SkyMatchStep attribute*), 81
reference_file_types (*romancal.skymatch.SkyMatchStep attribute*), 93
reference_file_types (*romancal.tweakreg.tweakreg_step.TweakRegStep attribute*), 71
reference_file_types (*romancal.tweakreg.TweakRegStep attribute*), 75
Region (*class in romancal.skymatch.region*), 91
remove_suffix() (*romancal.stpipe.RomanStep method*), 167
reproject() (*in module romancal.resample.resample_utils*), 120
resample_exposure_time() (*romancal.resample.resample.ResampleData method*), 117
resample_many_to_many() (*romancal.resample.resample.ResampleData method*), 117
resample_many_to_one() (*romancal.resample.resample.ResampleData method*), 117
resample_variance_array() (*romancal.resample.resample.ResampleData method*), 117
ResampleData (*class in romancal.resample.resample*), 114
ResampleStep (*class in romancal.resample*), 121
ResampleStep (*class in romancal.resample.resample_step*), 112
romancal.assign_wcs
    module, 44
romancal.dark_current
    module, 24
romancal.datamodels.container
    module, 177
romancal.dq_init
    module, 12
romancal.flatfield
    module, 48
romancal.flux
    module, 55
romancal.flux.flux_step
    module, 54
romancal.jump
    module, 29
romancal.lib.psf
    module, 62
romancal.linearity
    module, 21
romancal.outlier_detection
    module, 106
romancal.outlier_detection.outlier_detection
    module, 101
romancal.outlier_detection.outlier_detection_step
    module, 97
romancal.photom

module, 51
romancal.pipeline
module, 137
romancal.ramp_fitting
module, 39
romancal.resample
module, 121
romancal.resample.resample
module, 114
romancal.resample.resample_step
module, 112
romancal.resample.resample_utils
module, 118
romancal.saturation
module, 16
romancal.skymatch
module, 91
romancal.skymatch.region
module, 90
romancal.skymatch.skyimage
module, 84
romancal.skymatch.skymatch
module, 81
romancal.skymatch.skymatch_step
module, 80
romancal.skymatch.skystatistics
module, 89
romancal.source_detection
module, 60
romancal.stpipe
module, 166
romancal.tweakreg
module, 74
romancal.tweakreg.astrometric_utils
module, 72
romancal.tweakreg.tweakreg_step
module, 71
RomanPipeline (*class in romancal.stpipe*), 166
RomanStep (*class in romancal.stpipe*), 166

S

SaturationStep (*class in romancal.saturation*), 16
save() (*romancal.datamodels.container.ModelContainer method*), 179
scan() (*romancal.skymatch.region.Polygon method*), 90
scan() (*romancal.skymatch.region.Region method*), 91
set_buffer() (*romancal.datamodels.container.ModelContainer method*), 180
set_builtin_skystat() (*romancal.skymatch.skyimage.SkyImage method*), 88
set_data() (*romancal.skymatch.skyimage.DataAccessor method*), 84
set_data() (*romancal.skymatch.skyimage.NDArrayInMemoryAccessor method*), 84
set_data() (*romancal.skymatch.skyimage.NDArrayMappedAccessor method*), 84
set_drizzle_defaults() (*romancal.resample.resample_step.ResampleStep method*), 114
set_drizzle_defaults() (*romancal.resample.ResampleStep method*), 123
setup_output() (*romancal.pipeline.ExposurePipeline method*), 138
sky (*romancal.skymatch.skyimage.SkyGroup property*), 86
sky (*romancal.skymatch.skyimage.SkyImage property*), 88
SkyGroup (*class in romancal.skymatch.skyimage*), 85
SkyImage (*class in romancal.skymatch.skyimage*), 86
SkyMatchStep (*class in romancal.skymatch*), 92
SkyMatchStep (*class in romancal.skymatch.skymatch_step*), 80
skystat (*romancal.skymatch.skyimage.SkyImage property*), 88
SkyStats (*class in romancal.skymatch.skystatistics*), 89
SourceDetectionStep (*class in romancal.source_detection*), 60
spec (*romancal.dark_current.DarkCurrentStep attribute*), 25
spec (*romancal.flux.flux_step.FluxStep attribute*), 55
spec (*romancal.flux.FluxStep attribute*), 57
spec (*romancal.jump.JumpStep attribute*), 30
spec (*romancal.outlier_detection.outlier_detection_step.OutlierDetectionSt attribute*), 98
spec (*romancal.outlier_detection.OutlierDetectionStep attribute*), 107
spec (*romancal.pipeline.ExposurePipeline attribute*), 138
spec (*romancal.pipeline.HighLevelPipeline attribute*), 139
spec (*romancal.ramp_fitting.RampFitStep attribute*), 40
spec (*romancal.resample.resample_step.ResampleStep attribute*), 113
spec (*romancal.resample.ResampleStep attribute*), 122
spec (*romancal.skymatch.skymatch_step.SkyMatchStep attribute*), 81
spec (*romancal.skymatch.SkyMatchStep attribute*), 93
spec (*romancal.source_detection.SourceDetectionStep attribute*), 61
spec (*romancal.stpipe.RomanStep attribute*), 167
spec (*romancal.tweakreg.tweakreg_step.TweakRegStep attribute*), 71
spec (*romancal.tweakreg.TweakRegStep attribute*), 75
start (*romancal.skymatch.region.Edge property*), 90
step_defs (*romancal.pipeline.ExposurePipeline attribute*), 138

step_defs (*romancal.pipeline.HighLevelPipeline attribute*), 139
stop (*romancal.skymatch.region.Edge property*), 90

T

TweakRegStep (*class in romancal.tweakreg*), 74
TweakRegStep (*class in romancal.tweakreg.tweakreg_step*), 71

U

update_AET() (*romancal.skymatch.region.Polygon method*), 91
update_exposure_times() (*romancal.resample.resample.ResampleData method*), 117
update_phot_keywords() (*romancal.resample.resample_step.ResampleStep method*), 114
update_phot_keywords() (*romancal.resample.ResampleStep method*), 123

W

weighting (*romancal.ramp_fitting.RampFitStep attribute*), 41

Y

ymax (*romancal.skymatch.region.Edge property*), 90
ymin (*romancal.skymatch.region.Edge property*), 90