

---

**romancal**

***Release 0.8.2.dev6+gf86d8f2***

**Roman calibration pipeline developers**

**Sep 22, 2022**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Reference Files</b>	<b>5</b>
<b>3</b>	<b>CRDS</b>	<b>7</b>
<b>4</b>	<b>Running From the Command Line</b>	<b>9</b>
4.1	Exit Status . . . . .	9
<b>5</b>	<b>Running From the Python Prompt</b>	<b>11</b>
<b>6</b>	<b>Input and Output File Conventions</b>	<b>13</b>
6.1	Input Files . . . . .	13
6.2	Output Files . . . . .	13
<b>7</b>	<b>Parameters</b>	<b>15</b>
7.1	Universal Parameters . . . . .	15
7.2	Pipeline/Step Suffix Definitions . . . . .	16
<b>8</b>	<b>Package Documentation</b>	<b>17</b>
8.1	Package Index . . . . .	17
8.2	Error Propagation . . . . .	91
	<b>Python Module Index</b>	<b>93</b>
	<b>Index</b>	<b>95</b>



[genindex](#) | [modindex](#)



## **INTRODUCTION**

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





## 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 Teams. 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.



## 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/references/roman/
```



## RUNNING FROM THE COMMAND LINE

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

```
$ strun <pipeline_name, class_name, or 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 example, the Stage 1 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, `calroman.pipeline.ExposurePipeline` has the alias `roman_elp` and can be run as

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

### 4.1 Exit Status

`strun` produces the following exit status codes:

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



## RUNNING 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 more information, see [Execute via 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_uncal.asdf')
```

For more details on the different ways to run a pipeline step, see the [Configuring a Step](#) page.





## INPUT AND OUTPUT FILE CONVENTIONS

### 6.1 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.

### 6.2 Output Files

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

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

- The name of the input file
- 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.

#### 6.2.1 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 r0008308002010007027_06311_0019_WFI01_uncal.asdf
```

produces an output file named `r0008308002010007027_06311_0019_WFI01_dq_init.asdf`.

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



## 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 calroman.dq_init.DQInitStep -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'
```

### 7.1 Universal Parameters

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

#### 7.1.1 Output Directory

By default, all pipeline and step outputs will drop into the current working directory, i.e., the directory in which the process is running. To change this, use the `output_dir` parameter. 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
```

(continues on next page)

(continued from previous page)

```
--steps.dark_current.output_file='dark_sub.asdf'  
--steps.dark_current.output_dir='dark_calibrated'
```

## 7.1.2 Output File

When running a pipeline, the `stpipe` infrastructure automatically passes the output data model from one step to the input of the next step, without saving any intermediate results to disk.

## 7.2 Pipeline/Step 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	dq_init
Saturation detection	saturation
Linearity correction	linearity
Dark current	dark_current
Jump detection	jump
Corrected ramp data	rampfit
Optional fitting results from ramp_fit step	fitopt
Assign WCS	assign_wcs
Flat field	flat
Photometric calibration	phot
Calibrated image	cal

## PACKAGE DOCUMENTATION

### 8.1 Package Index

#### 8.1.1 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. The WCS object transforms positions in the detector frame to positions in the world coordinate frame - ICRS. The WCS 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 the 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 a frame `v2v3`. This part of the WCS pipeline may include intermediate coordinate frames. The basic WCS keywords are used to create the transform from frame `v2v3` to frame `world`.

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, [deg]

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

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

`RADESYS` - standard coordinate system [ICRS]

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

## Using the WCS interactively

Once a 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` object (as opposed to numbers) which allows further transformation of coordinates to different coordinate frames.

## romancal.assign\_wcs Package

### Classes

---

<code>AssignWcsStep</code> ([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 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 = ['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



## 8.1.2 Data Models

### 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 (TBD).

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.

## Working with models

### 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)
```

```
<class 'roman_datamodels.datamodels.ImageModel'>
```

Where the output of the type command tells you that you have imported an ImageModel from roman\_datamodels,

### Creating a data model from scratch

To create a new ImageModel, you can just:

```
from roman_datamodels import datamodels as rdm
from roman_datamodels.testing.factories import create_wfi_image
import numpy as np

image_node = create_wfi_image()
image_model = rdm.ImageModel(image_node)
type(image_model)
#<class 'roman_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.

### 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.

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



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

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

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

### Saving a data model to a file

Simply call the save method on the model instance. The format to save into will be deduced from the filename.:

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

**Note:** This save always clobbers the output file.

### 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 = old_model.copy()
```

### 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 im.items()), "\n")
```

which will list the contents of the ImageModel im:

```
meta.aperture.name Aperture name c1d861ddaebdb859f619fb2b79ea7bdf
meta.aperture.position_angle 115.33996998457596
meta.cal_step.flat_field SKIPPED

area <array (unloaded) shape: [4096, 4096] dtype: float32>
history.description HISTORY of this file
history.time 2021-12-29 14:03:57.465551
history.software.name roman_datamodels
history.software.author STSCI
history.software.homepage https://github.com/spacetelescope/roman_datamodels
history.software.version 0.8
```

or you can print specifics:

```
print("\n".join("{: >20}\t{}".format(k, v) for k, v in im.meta.wcsinfo.items()), "\n")
v2_ref 1312.9491452484797
v3_ref -1040.7853726755036
vparity -1
v3yangle -60.0
ra_ref 84.49289366006334
dec_ref -69.14101326380924
roll_ref 0.0
s_region NONE
```

Note: These will be incorporated as methods in the data models in a future release.

## 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.testing.factories import create_wfi_image
from roman_datamodels import datamodels as rdm
from romancal.datamodels import ImageModel
model = rdm.ImageModel(create_wfi_image())
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.

```
from roman_datamodels import datamodels as rdm
from roman_datamodels.testing.factories import create_wfi_science_raw
# Create a model of the desired type
raw = create_wfi_science_raw()
raw_science = rdm.ScienceRawModel(raw)
# find the associated Schema
raw_science.schema_uri
'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['observation']['start_time'])
print(raw_science.meta.observation.start_time)
```

In addition the times are stored as Astropy time objects and so the date can be displayed using various formats:

```
print(raw_science.meta.observation.start_time.iso)
2028-12-22 05:17:56.203
print(raw_science.meta.observation.start_time.mjd)
62127.22078938165
print(raw_science.meta.observation.start_time.yday)
2028:357:05:17:56.203
```

## Working with Roman datamodels and ASDF files

If you've 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:

(continues on next page)

(continued from previous page)

```

{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
<pre>rdm_a.meta rdm_a.meta.aperture rdm_a.meta.aperture.position_angle 120</pre>	<pre>asdf_a.tree['roman']['meta'] asdf_a.tree['roman']['meta']['aperture'] asdf_a.tree['roman']['meta']['aperture'] ↪'position_angle'] 120</pre>

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

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
```

## Using the info method

You can examine a roman data model using the `info` and `search` methods provided from the `asdf` package. The `info` function 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)
```

(continues on next page)

(continued from previous page)

```

├── [0] (ExtensionMetadata) ...
├── [1] (ExtensionMetadata) ...
├── [2] (ExtensionMetadata) ...
└── roman (WfiScienceRaw)
    ├── meta (dict)
    │   ├── aperture (Aperture) ...
    │   ├── cal_step (CalStep) ...
    │   ├── 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

```

import asdf
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 (CalStep)
│   ├── assign_wcs (str): INCOMPLETE
│   └── flat_field (str): INCOMPLETE
└── 6 not shown

```

(continues on next page)

(continued from previous page)

```

├─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 (CalStep)
├─assign_wcs (str)
├─flat_field (str)
├─dark (str)
├─dq_init (str)
├─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](#).

## 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 (CalStep)

```

or a general search for all attributes with `cal` in the name

```

>>> d_uncal.search('cal')
root (AsdfObject)
├─roman (WfiScienceRaw)
│   └─meta (dict)
│       ├──cal_step (CalStep)
│       ├──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 (CalStep)
└─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
│           └─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](#).

### 8.1.3 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. 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

---

<code><a href="#">DarkCurrentStep</a>([name, parent, config_file, ...])</code>	<code>DarkCurrentStep</code> : 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 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 = ['dark']
```

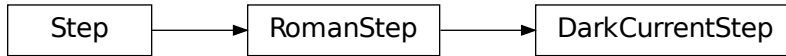
```
spec = '\n dark_output = output_file(default = None) # Dark corrected model\n '
```

### Methods Documentation

**process**(*input*)

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

## Class Inheritance Diagram



### 8.1.4 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.

## 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

---

<code>DQInitStep</code> ([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, initialize the `dq` attribute of the input model instead. The `dq` attribute of the MASK model is bitwise OR'd with the `pixeldq` (or `dq`) attribute of the input model.

Create a `Step` instance.

#### Parameters

- **name** (*str*, *optional*) – The name of the Step instance. Used in logging messages and in cache filenames. If not provided, one will be generated based on the class name.
- **parent** (*Step instance*, *optional*) – The parent step of this step. Used to determine a fully-qualified name for this step, and to determine the mode in which to run this step.
- **config\_file** (*str path*, *optional*) – The path to the config file that this step was initialized with. Use to determine relative path names of other config files.
- **\*\*kws** (*dict*) – 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 = ['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



## 8.1.5 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,

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

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

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

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

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

---

<code>FlatFieldStep</code> ( <i>name</i> , <i>parent</i> , <i>config_file</i> , ...)	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 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

---

<i>process</i> ( <i>step_input</i> )	This is where real work happens.
--------------------------------------	----------------------------------

---

### Attributes Documentation

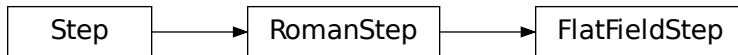
`reference_file_types = ['flat']`

## Methods Documentation

### `process(step_input)`

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

## Class Inheritance Diagram



## 8.1.6 Jump Detection

### Description

### Assumptions

We assume that the `saturation` step has already been applied to the input science exposure, so that saturated values are appropriately flagged in the input `GROUPDQ` array. We also assume that steps such as the reference pixel correction (`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 algorithm 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 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 elapsed time but at a slightly lower rate than the real cores.

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

## 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.

## 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

---

<code>JumpStep</code> ( <i>(name, parent, config_file, ...)</i> )	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 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 = ['gain', 'readnoise']
```

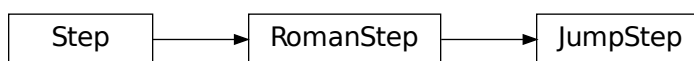
```
spec = "\n rejection_threshold = float(default=4.0,min=0) # CR sigma rej thresh\n three_group_rejection_threshold = float(default=6.0,min=0) # CR sigma rej thresh\n four_group_rejection_threshold = float(default=5.0,min=0) # CR sigma rej thresh\n maximum_cores = option('none', 'quarter', 'half', 'all', default='none') # max\n number of processes to create\n flag_4_neighbors = boolean(default=True) # flag the\n four perpendicular neighbors of each CR\n max_jump_to_flag_neighbors =\n float(default=1000) # maximum jump sigma that will trigger neighbor flagging\n min_jump_to_flag_neighbors = float(default=10) # minimum jump sigma that will\n trigger neighbor flagging\n "
```

### Methods Documentation

*process*(input)

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

### Class Inheritance Diagram



## 8.1.7 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  $n$ th-order polynomial for each pixel in the detector, with  $n+1$  arrays of coefficients read from the linearity reference file.

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

$$F_c = c_0 + c_1F + c_2F^2 + c_3F^3 + \dots + c_nF^n$$

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

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.instrument.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

---

<code>LinearityStep</code> ([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 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

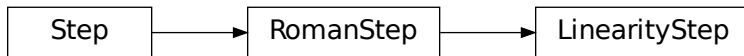
`reference_file_types = ['linearity']`

### 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



## 8.1.8 Pipeline Modules

### Pipeline Stages

End-to-end calibration of Roman data is divided into stages of processing:

- Stage 2 consists of detector-level corrections that are performed on a group-by-group basis, followed by ramp fitting. The output of stage 1 processing is a count rate image per exposure, or per integration for some modes. Details of this pipeline can be found at:
  - *exposure\_pipeline: Exposure Level Detector Processing*
  - Stage 2 processing consists of additional instrument-level and observing-mode corrections and calibrations to produce fully calibrated exposures. The details differ for imaging and spectroscopic exposures, and there are some corrections that are unique to certain instruments or modes. Details are at: TBD

The table below represents the same information as described above, but alphabetically ordered by pipeline class.

Pipeline Class   Alias	Used For
<a href="#">ExposurePipeline</a>   roman_elp	Stage 2:

## Pipelines and Exposure Type

The data from different observing modes needs to be processed with the proper pipeline stages listed above. The proper pipeline selection is usually based solely on the exposure type (`exposure.type` attribute).

### exposure\_pipeline: Exposure Level Detector 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
<i>dq_init</i>	✓	✓	✓
<i>saturation</i>	✓	✓	✓
<i>linearity</i>	✓	✓	✓
<i>dark_current</i>	✓	✓	✓
<i>jump</i>	✓	✓	✓
<i>ramp_fitting</i>	✓	✓	✓
<i>assign_wcs</i>	✓	✓	✓
<i>flatfield</i>	✓		

## Arguments

The exposure pipeline has one optional argument:

```
--save_calibrated_ramp boolean default=False
```

If set to `True`, the pipeline will save intermediate data to a file as it exists at the end of the `jump` step (just before ramp fitting). 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, including the detection and flagging of Cosmic-Ray (CR) hits within each ramp (integration). If created, the name of the intermediate file will be constructed from the root name of the input file, with the new product type suffix “`_ramp`” appended, e.g. “`r0008308002010007027_06311_0019_WFI01_ramp.asdf`”.

## 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 to zero.

## Outputs

### 2D Image model

**Data model**

ImageModel

**File suffix**

\_cal

Result of applying all pipeline steps up through the *flatfield* step, to produce corrected flatfield data which is 2D image data, which will have one less data dimensions as the input raw 3D data ( `ngroups x ncols x nrows` ). 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.

## 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.

---

## 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, and assign\_wcs. The flat field step is only applied to WFI imaging data.

See Step.\_\_init\_\_ for the parameters.

### Attributes Summary

---

*class\_alias*

---

*spec*

---

*step\_defs*

---

### Methods Summary

---

<i>create_fully_saturated_zeroed_image</i> (input_model)	Create zeroed-out image file
<i>process</i> (input)	Process the Roman WFI data
<i>setup_output</i> (input)	Determine the proper file name suffix to use later

---

### Attributes Documentation

```
class_alias = 'roman_elp'
```

```
spec = '\n save_calibrated_ramp = boolean(default=False)\n save_results =\n boolean(default=False)\n '
```

```
step_defs = {'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'>, 'jump': <class
'romancal.jump.jump_step.JumpStep'>, 'linearity': <class
'romancal.linearity.linearity_step.LinearityStep'>, 'phatom': <class
'romancal.phatom.phatom_step.PhotomStep'>, 'rampfit': <class
'romancal.ramp_fitting.ramp_fit_step.RampFitStep'>, 'saturation': <class
'romancal.saturation.saturation_step.SaturationStep'>}
```

## Methods Documentation

**create\_fully\_saturated\_zeroed\_image**(*input\_model*)

Create zeroed-out image file

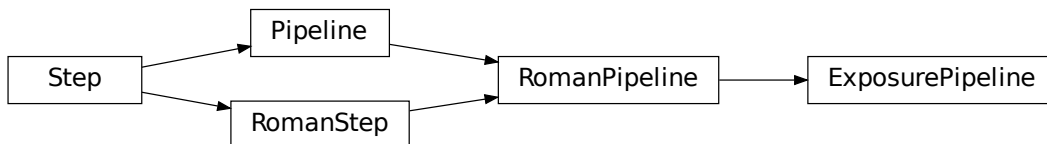
**process**(*input*)

Process the Roman WFI data

**setup\_output**(*input*)

Determine the proper file name suffix to use later

## Class Inheritance Diagram



## 8.1.9 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
	pixelarear	quantity	scalar	steradian

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

## romancal.photom Package

### Classes

---

<code>PhotomStep</code> ([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 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

---

<i>process</i> (input)	Perform the photometric calibration step
------------------------	--

---

### Attributes Documentation

**reference\_file\_types** = ['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

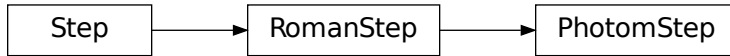
#### Returns

**output\_model** – output roman datamodel

#### Return type

Roman level 2 image datamodel (wfi\_image-1.x.x)

## Class Inheritance Diagram



### 8.1.10 Ramp Fitting

#### Description

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 elapsed 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:

$$\text{var}_o^C = \text{var}_o^R + \text{var}_o^P$$

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:

$$\text{slope}_o = \frac{\sum_s \frac{\text{slope}_s}{\text{var}_s^C}}{\sum_s \frac{1}{\text{var}_s^C}}$$

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.

## Arguments

The ramp fitting step has three optional arguments that can be set by the user:

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

## 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

## romancal.ramp\_fitting Package

### Classes

---

<i>RampFitStep</i> (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 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

---

*algorithm*

---

*reference\_file\_types*

---

*spec*

---

*weighting*

---

#### Methods Summary

---

*process*(input)

This is where real work happens.

---

## Attributes Documentation

```

algorithm = 'ols'

reference_file_types = ['readnoise', 'gain']

spec = "\n opt_name = string(default='')\n maximum_cores =
option('none','quarter','half','all',default='none') # max number of processes to
create\n save_opt = boolean(default=False) # Save optional output\n "

weighting = 'optimal'

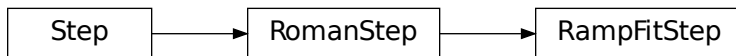
```

## Methods Documentation

`process(input)`

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

## Class Inheritance Diagram



## 8.1.11 Reference File Information

### Introduction

This document is intended to be a core reference guide to the formats, naming convention and data quality flags used by the reference files for pipeline steps requiring them, and is not intended to be a detailed description of each of those pipeline steps. It also does not give details on pipeline steps that do not use reference files. The present manual is the living document for the reference file specifications.

### 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”.

## 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)
<i>assign_wcs</i>	DISTORTION
<i>dark_current</i>	<i>DARK</i>
<i>dq_init</i>	<i>MASK</i>
<i>flatfield</i>	<i>FLAT</i>
<i>jump_detection</i>	GAIN
	READNOISE
<i>linearity</i>	LINEARITY
<i>photom</i>	<i>PHOTOM</i>
<i>ramp_fitting</i>	GAIN
	READNOISE
<i>saturation</i>	<i>SATURATION</i>

Reference File Type (reftype)	Pipeline Step
<i>DARK</i>	<i>dark_current</i>
DISTORTION	<i>assign_wcs</i>
<i>FLAT</i>	<i>flatfield</i>
GAIN	<i>jump_detection</i>
	<i>ramp_fitting</i>
LINEARITY	<i>linearity</i>
<i>MASK</i>	<i>dq_init</i>
<i>PHOTOM</i>	<i>photom</i>
READNOISE	<i>jump_detection</i>
	<i>ramp_fitting</i>
<i>SATURATION</i>	<i>saturation</i>

## Standard ASDF metadata

All Roman science and reference files are ASDF files.

The required attributes Documenting Contents of Reference Files are:



At-tribute	Comment
reftype	FLAT Required values are listed in the discussion of each pipeline step.
de-scrip-tion	Summary of file content and/or reason for delivery.
au-thor	Fred Jones Person(s) who created the file.
use-after	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.
pedi-gree	Options are 'SIMULATION' 'GROUND' 'DUMMY' 'INFLIGHT YYYY-MM-DD YYYY-MM-DD'
his-tory	Description of Reference File Creation.
tele-scope	ROMAN Name of the telescope/project.
in-stru-ment	WFI Instrument name.

### 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'.

## 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	RESERVED_1	
5	32	PERSISTENCE	High persistence (was RESERVED_2)
6	64	AD_FLOOR	Below A/D floor (0 DN, was RESERVED_3)
7	128	RESERVED_4	
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

### 8.1.12 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), 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.

## 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

---

<code>SaturationStep([name, parent, config_file, ...])</code>	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 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

---

<i>process</i> (input)	This is where real work happens.
------------------------	----------------------------------

---

### Attributes Documentation

**reference\_file\_types** = ['saturation']

### Methods Documentation

**process**(*input*)

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

## Class Inheritance Diagram



### 8.1.13 STPIPE

#### For Users

#### Steps

#### Configuring a Step

This section describes how to instantiate a Step and set configuration parameters on it.

Steps can be configured by:

- 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 be 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
```

(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.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:
```

(continues on next page)

(continued from previous page)

```
- 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:
  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
```

(continues on next page)

(continued from previous page)

```
[*.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>
```

(continues on next page)

```
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
  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')
```

(continues on next page)

(continued from previous page)

```
# 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*.

## 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")

        # 2.
        threshold = self.threshold

        # 3.
        with ImageModel(image1) as image1, ImageModel(image2) as image2:
            # 4.
            with self.get_reference_file_model(image1, "flat_field") as flat:
                new_image = combine(image1, image2, flat, threshold)

        # 5.
        return new_image

    # 6.
    spec = """
    # This is the configspec file for ExampleStep

    threshold = float(default=1.0) # maximum flux
    """

    # 7.
    reference_file_types = ['flat_field']

```

The Python Step subclass may be installed anywhere that your Python installation can find it. It does not need to be installed in the stpipe package.

### The spec member

The spec member variable is a string containing information about the parameters. It is in the configspec format defined in the ConfigObj library that stpipe uses.

The configspec format defines the types of the parameters, as well as allowing an optional tree structure.

The types of parameters are declared like this:

```

n_iterations = integer(1, 100) # The number of iterations to run
factor = float() # A multiplication factor
author = string() # The author of the file

```



Note that each parameter may have a comment. This comment is extracted and displayed in help messages and docstrings etc.

Parameters can be grouped into categories using ini-file-like syntax:

```
[red]
offset = float()
scale = float()

[green]
offset = float()
scale = float()

[blue]
offset = float()
scale = float()
```

Default values may be specified on any parameter using the `default` keyword argument:

```
name = string(default="John Doe")
```

While the most commonly useful parts of the configspec format are discussed here, greater detail can be found in the [configspec documentation](#).

## 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.
    """

    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()

# 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

---

<code>RomanPipeline(*args, **kwargs)</code>	See <code>Step.__init__</code> for the parameters.
<code>RomanStep([name, parent, config_file, ...])</code>	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 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

<i>finalize_result</i> (model, reference_files_used)	Hook that allows the Step to set metadata on the output model before save.
<i>record_step_status</i> (model, step_name[, success])	Record step completion status in the model's metadata.
<i>remove_suffix</i> (name)	Remove any Roman step-specific suffix from the given filename.

### Attributes Documentation

```
spec = "\n output_ext = string(default='.asdf') # Default type of output\n "
```

## Methods Documentation

**finalize\_result**(*model*, *reference\_files\_used*)

Hook that allows the Step to set metadata on the output model before save.

### Parameters

- **model** (*stdatamodels.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** (*stdatamodels.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.

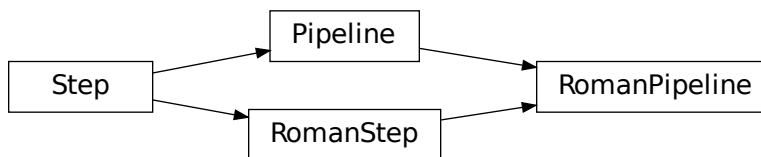
### Returns

Filename with step suffix removed.

### Return type

*str*

## Class Inheritance Diagram



## 8.1.14 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 during `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.

## 8.2 Error Propagation

### 8.2.1 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
<code>ramp_fitting</code>	ELPP	VAR_POISSON, VAR_RNOISE	ERR	None
<code>flat_field</code>	ELPP	VAR_FLAT	ERR, VAR_POISSON, VAR_RNOISE	None

## 8.2.2 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](#).



## PYTHON MODULE INDEX

### r

`romancal.assign_wcs`, 18  
`romancal.dark_current`, 30  
`romancal.dq_init`, 34  
`romancal.flatfield`, 38  
`romancal.jump`, 41  
`romancal.linearity`, 45  
`romancal.photom`, 52  
`romancal.pipeline`, 48  
`romancal.ramp_fitting`, 58  
`romancal.saturation`, 64  
`romancal.stpipe`, 88



## A

algorithm (*romancal.ramp\_fitting.RampFitStep* attribute), 59

AssignWcsStep (*class in romancal.assign\_wcs*), 18

## C

class\_alias (*romancal.pipeline.ExposurePipeline* attribute), 49

create\_fully\_saturated\_zeroed\_image() (*romancal.pipeline.ExposurePipeline* method), 50

## D

DarkCurrentStep (*class in romancal.dark\_current*), 31

DQInitStep (*class in romancal.dq\_init*), 34

## E

ExposurePipeline (*class in romancal.pipeline*), 49

## F

finalize\_result() (*romancal.stpipe.RomanStep* method), 90

FlatFieldStep (*class in romancal.flatfield*), 38

## J

JumpStep (*class in romancal.jump*), 41

## L

LinearityStep (*class in romancal.linearity*), 45

## M

module

- romancal.assign\_wcs, 18
- romancal.dark\_current, 30
- romancal.dq\_init, 34
- romancal.flatfield, 38
- romancal.jump, 41
- romancal.linearity, 45
- romancal.photom, 52
- romancal.pipeline, 48
- romancal.ramp\_fitting, 58
- romancal.saturation, 64

romancal.stpipe, 88

## P

PhotomStep (*class in romancal.photom*), 52

process() (*romancal.assign\_wcs.AssignWcsStep* method), 19

process() (*romancal.dark\_current.DarkCurrentStep* method), 31

process() (*romancal.dq\_init.DQInitStep* method), 35

process() (*romancal.flatfield.FlatFieldStep* method), 39

process() (*romancal.jump.JumpStep* method), 42

process() (*romancal.linearity.LinearityStep* method), 46

process() (*romancal.photom.PhotomStep* method), 53

process() (*romancal.pipeline.ExposurePipeline* method), 50

process() (*romancal.ramp\_fitting.RampFitStep* method), 59

process() (*romancal.saturation.SaturationStep* method), 65

## R

RampFitStep (*class in romancal.ramp\_fitting*), 58

record\_step\_status() (*romancal.stpipe.RomanStep* method), 90

reference\_file\_types (*romancal.assign\_wcs.AssignWcsStep* attribute), 19

reference\_file\_types (*romancal.dark\_current.DarkCurrentStep* attribute), 31

reference\_file\_types (*romancal.dq\_init.DQInitStep* attribute), 35

reference\_file\_types (*romancal.flatfield.FlatFieldStep* attribute), 38

reference\_file\_types (*romancal.jump.JumpStep* attribute), 42

reference\_file\_types (*romancal.linearity.LinearityStep* attribute), 46

reference\_file\_types (*romancal.photom.PhotomStep* attribute), 53

reference\_file\_types (*romancal.ramp\_fitting.RampFitStep* attribute), 59

reference\_file\_types (*romancal.saturation.SaturationStep* attribute), 65

remove\_suffix() (*romancal.stpipe.RomanStep* method), 90

romancal.assign\_wcs module, 18

romancal.dark\_current module, 30

romancal.dq\_init module, 34

romancal.flatfield module, 38

romancal.jump module, 41

romancal.linearity module, 45

romancal.photom module, 52

romancal.pipeline module, 48

romancal.ramp\_fitting module, 58

romancal.saturation module, 64

romancal.stpipe module, 88

RomanPipeline (*class in romancal.stpipe*), 88

RomanStep (*class in romancal.stpipe*), 89

## S

SaturationStep (*class in romancal.saturation*), 65

setup\_output() (*romancal.pipeline.ExposurePipeline* method), 50

spec (*romancal.dark\_current.DarkCurrentStep* attribute), 31

spec (*romancal.jump.JumpStep* attribute), 42

spec (*romancal.pipeline.ExposurePipeline* attribute), 49

spec (*romancal.ramp\_fitting.RampFitStep* attribute), 59

spec (*romancal.stpipe.RomanStep* attribute), 89

step\_defs (*romancal.pipeline.ExposurePipeline* attribute), 49

## W

weighting (*romancal.ramp\_fitting.RampFitStep* attribute), 59