romanisim

Release 0.5.1.dev14+g6a98ffb.d20240503

STScI <help@stsci.edu>

May 03, 2024

CONTENTS

A Roman WFI image simulator based on galsim.

We stylize the simulator Roman I-Sim and pronounce it roman - eye - sim; the package is romanisim.

CHAPTER

ONE

CONTENTS

1.1 Overview

romanisim simulates Roman Wide-Field Imager images of astronomical scenes described by catalogs of sources. The simulation includes:

- convolution of the sources by the Roman point spread function
- optical distortion
- sky background
- level 1 image support (3D image stack of up-the-ramp samples)
- level 2 image support (2D image after calibration & ramp fitting)
- point sources and analytic galaxy profiles
- expected system throughput
- dark current
- read noise
- inter-pixel capacitance
- non-linearity
- saturation
- ramp fitting
- source injection

The simulator is based on galsim and most of these features directly invoke the equivalents in the galsim.roman package. The chief additions of this package on top of the galsim.roman implementation are using "official" webbpsf PSF, and distortion and reference files from the Roman CRDS (not yet public!). This package also implements WFI up-the-ramp sampled and averaged images like those that will be downlinked from the telescope, and the official Roman WFI file format (asdf).

Future expected features include:

- frame zero effects
- L3 image simulations
- "image-based" simulation inputs (i.e., provide an input image based on a galaxy hydro sim; romanisim applies the Roman PSF & instrumental effects on top to produce a detailed instrumental simulation)

The best way to interact with romanisim is to make an image. Running

```
romanisim-make-image out.asdf
```

will make a test image in the file `out.asdf`. Naturally, usually one has a particular astronomical scene in mind, and one can't really simulate a scene without knowing where the telescope is pointing and when the observation is being made. A more complete invocation would be

```
romanisim-make-image --catalog input.ecsv --radec 270 66 --bandpass F087 --sca 7 --date␣
↪2026 1 1 --level 1 out.asdf
```

where `input.ecsv` includes a list of sources in the scene, the telescope boresight is pointing to (r, d) = (270, 66), the desired bandpass is F087, the sensor is WFI07, the date is Jan 1, 2026, and a level 1 image (3D cube of samples up the ramp) is requested.

The output of `romanisim-make-image` is an appropriate asdf file for the requested level image, with the following addition. The script adds an additional top-level branch to the asdf tree with the name `romanisim`. Here's an example:

```
└─romanisim (dict)
  ├─bandpass (str): F087
  ├─catalog (NoneType): None
  ├─date (NoneType): None
  ├─filename (str): out.asdf
  ├─level (int): 1
  ├─ma_table_number (int): 1
  ├─radec (NoneType): None
  ├─sca (int): 7
  ├─rng_seed (NoneType): None
  ├─simcatobj (NDArrayType): shape=(496,), dtype=void96
  ├─usecrds (bool): False
  └─webbpsf (bool): True
```

These fields are simply the arguments to `romanisim-make-image`, plus an additional `simcatobj` field with contains the `x`, `y`, and number of photons of each simulated source.

Features not included so far:

- pedestal/frame 0 features
- non-linear dark features

1.2 Installation

To install

```
pip install romanisim
```

and you should be largely set!

There are a few dependencies that may cause more difficulty. First, WebbPSF requires data files to operate. See the docs for instructions on obtaining the relevant data files and pointing the `WEBBPSF_PATH` environment variable to them. This issue can be avoided by not setting the `--webbpsf` argument, in which case `romanisim` uses the GalSim modeling of the Roman PSF.

Second, some synthetic scene generation tools use images of galaxies distributed separately from the main GalSim source. See here for information on obtaining the COSMOS galaxies for use with GalSim. The `romanisim` package also has a less sophisticated scene modeling toolkit, which just renders Sersic galaxies. The command line interface

to romanisim presently uses supports Sersic galaxy rendering, and so many users may not need to download the COSMOS galaxies.

Third, romanisim can work with the Roman CRDS system. This functionality is not available to the general community at the time of writing. Using CRDS requires specifying the CRDS_PATH and CRDS_SERVER_URL variables. CRDS is not used unless the --usecrds argument is specified; do not include this argument unless you have access to the Roman CRDS.

That said, the basic install process looks like this:

```
pip install romanisim
# to get a specific version, use instead
# pip install romanisim==0.1
# to be able to run the tests for a specific version, use instead
# pip install romanisim[test]==0.1

# get webbpsf data and untar it
mkdir -p $HOME/data/webbpsf-data
cd $HOME/data/webbpsf-data
wget https://stsci.box.com/shared/static/qxpiaxsjwo15ml6m4pkhtk36c9jgj70k.gz -O webbpsf-
↪data.tar.gz
tar -xzf webbpsf-data.tar.gz
export WEBBPSF_PATH=$PWD/webbpsf-data

# get galsim galaxy catalogs
# Note: ~5 GB each, takes a little while to download.
# Both are needed for tests.  Neither are needed if you are
# exclusively using analytic model galaxies.
galsim_download_cosmos -s 23.5
galsim_download_cosmos -s 25.2
```

You may wish to, for example, set up a new python virtual environment before running the above, or choose a different directory for WebbPSF's data files.

Some users report issues with the FFTW dependency of galsim on Mac Arm systems. See galsim's installation page for hints there. In particular it may be helpful to install FFTW before galsim and romanisim.

1.3 Running the simulation

The primary means by which we expect most users to make images is the command line interface:

```
romanisim-make-image out.asdf
```

The combination of romanisim-make-image and various user-generated input catalogs allows most simulator functionality to be exercised[1].

The romanisim-make-image CLI has a number of arguments to support this functionality:

```
romanisim-make-image -h
usage: romanisim-make-image [-h] [--catalog CATALOG] [--radec RADEC RADEC] [--bandpass␣
↪BANDPASS]
                            [--sca SCA] [--usecrds] [--webbpsf] [--date DATE [DATE ...]]
```

(continues on next page)

[1] An important exception is the chromatic PSF rendering and photon-shooting modes of GalSim; the current catalog format does not support chromatic PSF rendering, and just assumes that all sources are "gray" within a bandpass.

(continued from previous page)

```
                          [--level LEVEL] [--ma_table_number MA_TABLE_NUMBER] [--seed␣
↪SEED]
                          [--nobj NOBJ] [--boresight] [--previous PREVIOUS]
                          filename

Make a demo image.

positional arguments:
  filename              output image (fits)

optional arguments:
  -h, --help            show this help message and exit
  --catalog CATALOG     input catalog (csv) (default: None)
  --radec RADEC RADEC   ra and dec (deg) (default: None)
  --bandpass BANDPASS   bandpass to simulate (default: F087)
  --sca SCA             SCA to simulate (default: 7)
  --usecrds             Use CRDS for distortion map (default: False)
  --webbpsf             Use webbpsf for PSF (default: False)
  --date DATE [DATE ...]
                        Date of observation to simulate: year month day hour minute␣
↪second
                        microsecond (default: None)
  --level LEVEL         1 or 2, for L1 or L2 output (default: 2)
  --ma_table_number MA_TABLE_NUMBER
  --rng_seed SEED
  --nobj NOBJ
  --boresight           radec specifies location of boresight, not center of WFI.␣
↪(default: False)
  --previous PREVIOUS   previous simulated file in chronological order used for␣
↪persistence modeling.
                        (default: None)

EXAMPLE: romanisim-make-image output_image.fits
```

Expected arguments controlling things like the input *here* to simulate, the right ascension and declination of the telescope[2], the *bandpass*, the SCA to simulate, the level of the image to simulate (*L1* or *L2*), the MA table to use, and the time of the observation.

Additional arguments control some details of the simulation. The `--usecrds` argument indicates that reference files should be pulled from the Roman CRDS server; this is the recommended option when CRDS is available. The `--webbpsf` argument indicates that the WebbPSF package should be used to simulate the PSF; note that this presently disables chromatic PSF rendering.

The `--rng_seed` argument specifies a seed to the random number generator, enabling reproducible results.

The `--nobj` argument is only used when a catalog is not specified, and controls the number of objects that are simulated in that case.

The `previous` argument specifies the previous simulated frame. This information is used to support *persistence* modeling.

[2] This right ascension corresponds to either the location of the center of the WFI array or the telescope boresight, when the `--boresight` argument is specified.

1.4 Making images

Ultimately, romanisim builds image by feeding source profiles, world coordinate system objects, and point spread functions to galsim. The `image` and `l1` modules currently implement this functionality.

The `image` module is responsible for translating a metadata object that specifies everything about the conditions of the observation into objects that the simulation can understand. The metadata object follows the metadata that real WFI images will include; see here for more information.

The parsed metadata is used to make a `counts` image that is an idealized image containing the number of photons each WFI pixel would collect over an observation. It includes no systematic effects or noise beyond Poisson noise from the astronomical scene and backgrounds. Actual WFI observations are more complicated than just noisy versions of this idealized image, however, for several reasons:

- WFI pixels have a very uncertain pedestal.
- WFI pixels are sampled "up the ramp" during an observation, so a number of reads contribute to the final estimate for the rate of photons entering each pixel.
- WFI reads are averaged on the telescope into resultants; ground images see only resultants.

These idealized count images are then used to either make a level 2 image or a level 1 image, which are intended to include the effects of these complications. The construction of L1 images is described *here*, and the construction of L2 images is described *here*.

1.4.1 romanisim.image Module

Roman WFI simulator tool.

Based on galsim's implementation of Roman image simulation. Uses galsim Roman modules for most of the real work.

Functions

add_objects_to_image(image, objlist, xpos, ...)	Add sources to an image.
gather_reference_data(image_mod[, usecrds])	Gather reference data corresponding to metadata.
in_bounds(xx, yy, imbd, margin)	Filter sources to those landing on an image.
make_asdf(slope, slopevar_rn, slopevar_poisson)	Wrap a galsim simulated image with ASDF/roman_datamodel metadata.
make_l2(resultants, read_pattern[, ...])	Simulate an image in a filter given resultants.
make_test_catalog_and_images([seed, sca, ...])	This routine kicks the tires on everything in this module.
simulate(metadata, objlist[, usecrds, ...])	Simulate a sequence of observations on a field in different bandpasses.
simulate_counts(metadata, objlist[, rng, ...])	Simulate total counts in a single SCA.
simulate_counts_generic(image, exptime[, ...])	Add some simulated counts to an image.
trim_objlist(objlist, image)	Trim a Table of objects down to those falling near an image.

add_objects_to_image

romanisim.image.**add_objects_to_image**(*image*, *objlist*, *xpos*, *ypos*, *psf*, *flux_to_counts_factor*, *bandpass=None*, *filter_name=None*, *rng=None*, *seed=None*)

Add sources to an image.

Note: this includes Poisson noise when photon shooting is used (i.e., for chromatic source profiles), and otherwise is noise free.

Parameters

image
[galsim.Image] Image to which to add sources with associated WCS.

objlist
[list[CatalogObject]] Objects to add to image

xpos, ypos
[array_like] x & y positions of sources (pixel) at which sources should be added

psf
[galsim.Profile] PSF for image

flux_to_counts_factor
[float] physical fluxes in objlist (whether in profile SEDs or flux arrays) should be multiplied by this factor to convert to total counts in the image

bandpass
[galsim.Bandpass] bandpass in which image is being rendered. This is used only in cases where chromatic profiles & PSFs are being used.

filter_name
[str] filter to use to select appropriate flux from objlist. This is only used when achromatic PSFs and sources are being rendered.

rng
[galsim.BaseDeviate] random number generator to use

seed
[int] seed to use for random number generator

Returns

outinfo
[np.ndarray] Array structure containing rows for each source. The columns give the total number of counts from the source entering the image and the time taken to render the source.

gather_reference_data

romanisim.image.**gather_reference_data**(*image_mod*, *usecrds=False*)

Gather reference data corresponding to metadata.

This function pulls files from parameters.reference_data and/or CRDS to fill out the various reference files needed to perform the simulation. If CRDS is set, values in parameters.reference_data are used instead of CRDS files when the reference_data are None. If all CRDS files should be used, parameters.reference_data must contain only Nones.

This functionality is intended to allow users to specify different levels via a configuration file and not have them be overwritten by the CRDS defaults, but it's not terribly clean.

The input metadata is updated with CRDS software versions if CRDS is used.

Returns

dictionary containing the following keys:
read_noise darkrate gain inv_linearity linearity saturation reffiles

These have the reference images or constant values for the various reference parameters.

in_bounds

romanisim.image.**in_bounds**(*xx*, *yy*, *imbd*, *margin*)

Filter sources to those landing on an image.

Parameters

xx, yy: ndarray[nobj] (float)
x & y positions of sources on image

imbd
[galsim.Image.Bounds] bounds of image

margin
[int] keep sources up to margin outside of bounds

Returns

keep
[np.ndarray (bool)] whether each source lands near the image (True) or not (False)

make_asdf

romanisim.image.**make_asdf**(*slope*, *slopevar_rn*, *slopevar_poisson*, *metadata=None*, *filepath=None*, *persistence=None*, *dq=None*, *imwcs=None*)

Wrap a galsim simulated image with ASDF/roman_datamodel metadata.

Eventually this needs to get enough info to reconstruct a refit WCS.

make_l2

romanisim.image.**make_l2**(*resultants*, *read_pattern*, *read_noise=None*, *gain=None*, *flat=None*, *linearity=None*, *darkrate=None*, *dq=None*)

Simulate an image in a filter given resultants.

This routine does idealized ramp fitting on a set of resultants.

Parameters

resultants
[np.ndarray[nresultants, nx, ny]] resultants array

read_pattern
[list[list] (int)] list of list of indices of reads entering each resultant

read_noise
[np.ndarray[nx, ny] (float)] read_noise image to use. If None, use galsim.roman.read_noise.

flat
[np.ndarray[nx, ny] (float)] flat field to use

linearity
[romanisim.nonlinearity.NL object or None] non-linearity correction to use.

darkrate
[np.ndarray[nx, ny] (float)] dark rate image to subtract from ramps (electron / s)

dq
[np.ndarray[nresultants, nx, ny] (int)] DQ image corresponding to resultants

Returns

im
[np.ndarray] best fitting slopes

var_rnoise
[np.ndarray] variance in slopes from read noise

var_poisson
[np.ndarray] variance in slopes from source noise

make_test_catalog_and_images

romanisim.image.**make_test_catalog_and_images**(*seed=12345*, *sca=7*, *filters=None*, *nobj=1000*, *usecrds=True*, *webbpsf=True*, *galaxy_sample_file_name=None*, ***kwargs*)

This routine kicks the tires on everything in this module.

simulate

romanisim.image.**simulate**(*metadata*, *objlist*, *usecrds=True*, *webbpsf=True*, *level=2*, *crparam={}*, *persistence=None*, *seed=None*, *rng=None*, *psf_keywords={}*, ***kwargs*)

Simulate a sequence of observations on a field in different bandpasses.

Parameters

metadata
[dict] metadata structure for Roman asdf file, including information about

- pointing: metadata['wcsinfo']['ra_ref'], metadata['wcsinfo']['dec_ref']
- date: metadata['exposure']['start_time']
- sca: metadata['instrument']['detector']
- bandpass: metadata['instrument']['optical_detector']
- ma_table_number: metadata['exposure']['ma_table_number']

objlist
[list[CatalogObject] or Table] List of objects in the field to simulate

usecrds
[bool] use CRDS to get distortion maps

webbpsf
[bool] use webbpsf to generate PSF

level
[int] 0, 1 or 2, specifying level 1 or level 2 image 0 makes a special idealized 'counts' image

persistence
[romanisim.persistence.Persistence] persistence object to use; None for no persistence

crparam
[dict] Parameters for cosmic ray simulations. None for no cosmic rays. Empty dictionary for default parameters.

rng
[galsim.BaseDeviate] Random number generator to use

seed
[int] Seed for populating RNG. Only used if rng is None.

psf_keywords
[dict] Keywords passed to the PSF generation routine

Returns

image
[roman_datamodels model] simulated image

extras
[dict] Dictionary of additionally valuable quantities. Includes at least simcatobj, the image positions and fluxes of simulated objects. It may also include information on persistence and cosmic ray hits.

simulate_counts

romanisim.image.**simulate_counts**(*metadata*, *objlist*, *rng=None*, *seed=None*, *ignore_distant_sources=10*, *usecrds=True*, *webbpsf=True*, *darkrate=None*, *flat=None*, *psf_keywords={}*)

Simulate total counts in a single SCA.

This gives the total counts in an idealized instrument with no systematics; it includes only distortion & PSF convolution.

Parameters

metadata
[dict] CRDS metadata dictionary

objlist
[list[CatalogObject] or Table] Objects to simulate

rng
[galsim.BaseDeviate] Random number generator to use

seed
[int] Seed for populating RNG. Only used if rng is None.

ignore_distant_sources
[float] do not render sources more than this many pixels off edge of detector

usecrds
[bool] use CRDS distortion map

darkrate
[float or np.ndarray[float]] dark rate image to use (electrons / s)

flat
[float or np.ndarray[float]] flat field to use

psf_keywords
[dict] keywords passed to PSF generation routine

Returns

image
[galsim.Image] idealized image of scene as seen by Roman, giving total electron counts from rate sources (astronomical objects; backgrounds; dark current) in each pixel.

simcatobj
[np.ndarray] catalog of simulated objects in image

simulate_counts_generic

romanisim.image.**simulate_counts_generic**(*image*, *exptime*, *objlist=None*, *psf=None*, *zpflux=None*, *sky=None*, *dark=None*, *flat=None*, *xpos=None*, *ypos=None*, *ignore_distant_sources=10*, *bandpass=None*, *filter_name=None*, *rng=None*, *seed=None*)

Add some simulated counts to an image.

No Roman specific code allowed! To do this, we need to have an image to start with with an attached WCS. We also need an exposure time and potentially a zpflux so we know how to translate between the catalog fluxes and the counts entering the image. For chromatic rendering, this role instead is played by the bandpass, though the exposure time is still needed to handle that part of the conversion from flux to counts.

Then there are a few of individual components that can be added on to an image:

- objlist: a list of CatalogObjects to render, or a Table. Can be chromatic or not. This will have all your normal PSF and galaxy profiles.
- sky: a sky background model. This is different from a dark in that it is sensitive to the flat field.
- dark: a dark model.
- flat: a flat field for modulating the object and sky counts

Parameters

image
[galsim.Image] Image onto which other effects should be added, with associated WCS.

exptime
[float] Exposure time

objlist
[list[CatalogObject], Table, or None] Sources to render

psf
[galsim.Profile] PSF to use when rendering sources

zpflux
[float] For non-chromatic profiles, the factor converting flux to counts / s.

sky
[float or array_like] Image or constant with the counts / pix / sec from sky.

dark
[float or array_like] Image or constant with the counts / pix / sec from dark current.

flat
[array_like] Image giving the relative QE of different pixels.

xpos, ypos
[array_like (float)] x, y positions of each source in objlist

ignore_distant_sources
[int] Ignore sources more than this distance off image.

bandpass
[galsim.Bandpass] bandpass to use for rendering chromatic objects

filter_name
[str] name of filter (used to look up flux in achromatic case)

rng
[galsim.BaseDeviate] random number generator

seed
[int] seed for random number generator

Returns

objinfo
[np.ndarray] Information on position and flux of each rendered source.

trim_objlist

romanisim.image.**trim_objlist**(*objlist*, *image*)

Trim a Table of objects down to those falling near an image.

Objects must fall in a circle centered at the center of the image with radius 1.1 times the separation between the center and corner of the image.

In contrast to in_bounds, this doesn't require the x and y coordinates of the sources, and just uses the ra/dec directly without needing to do the WCS transformation.

Parameters

objlist
[astropy.table.Table including ra, dec columns] Table of objects

image
[galsim.Image] image near which objects should fall.

Returns

objlist
[astropy.table.Table] objlist trimmed to objects near image.

1.5 Making L1 images

An L1 (level 1) image is a "raw" image received from the detectors. The actual measurements made on the spacecraft consist of a number of non-destructive reads of the pixels of the H4RG detectors. These reads have independent read noise but because the pixels count the total number of photons having entered each pixel, the Poisson noise in different reads of the same pixel is correlated.

Because the telescope has limited bandwidth, every read is not transferred to ground stations. Instead, reads are averaged into "resultants" according to a specification called a MultiAccum table, and these resultants are transferred, archived, and analyzed. These resultants make up an L1 image, which romanisim simulates.

L1 images are created using an idealized `counts` image described *here*, which contains the number of photons each pixel of the detector would receive absent any instrumental systematics. To transform this into an L1 image, these counts must be apportioned into reads and averaged into resultants, and instrumental effects must be added.

This process proceeds by simulating each read, drawing the appropriate number of photons from the total number of photons for each read following a binomial distribution. These photons are added to a running sum that is then averaged into a resultant according to the MultiAccum table specification. This process requires drawing random numbers from the binomial distribution for every read of every pixel, and so can take on the order of a minute, but it allows detailed simulation of the statistics of the noise in each resultant together with their correlations. It also makes it straightforward to add various instrumental effects into the simulation accurately, since these usually apply to individual reads rather than to resultants (e.g., cosmic rays affect individual reads, and their affect on a resultant depends on the read in the resultant to which they apply).

After apportioning counts to resultants, systematic effects are added to the resultants. Presently only read noise is added. The read noise is averaged down like $1/\sqrt{N}$, where N is the number of reads contributing to the resultant.

1.5.1 Nonlinearity

Non-linearity is considered when L1 images are constructed and a non-linearity model is provided (e.g., from CRDS). We treat non-linearity as a difference between the electrons captured in the detector and the amount of signal read out. This function is modeled as a high order polynomial, and the coefficients of this polynomial and its inverse are stored in CRDS (linearity, inverselinearity reference files for each detector). When assigning counts to each read, these are transformed through the inverselinearity polynomial for each pixel and then added to the resultant buffer to account for this effect. The linearity polynomial then corrects for this effects as part of calibrating an L1 file and eventually producing an L2 file.

The linearity polynomials are occasionally poorly determined or cannot be computed. When marked as problematic in the reference file, we use trivial polynomials (i.e., the identity), and mark the affected pixels with a DQ bit indicating a problematic linearity correction.

1.5.2 Interpixel Capacitance

Interpixel capacitance (IPC) is added following non-linearity and before read-out. Read noise remains independent among different pixels but the Poisson noise is correlated between pixels by the IPC. We simply convolve the resultants by a 3x3 kernel after apportioning counts to resultants and applying non-linearity but before adding read noise.

This is slightly different than including IPC in the PSF kernel because including IPC in the PSF kernel leaves the Poisson noise uncorrelated.

1.5.3 Persistence

Persistence is implemented in the simulator following Sanchez+2023. This follows the Fermi description of persistence implemented in GalSim, where the flux in electrons per second recorded in a pixel is parameterized in terms of the total number of counts recorded in an earlier frame.

$$P(t) = A\frac{1}{1+\exp\left(\frac{-(x-x_0)}{\delta x}\right)}\left(\frac{x}{x_0}\right)^{\alpha}\left(\frac{t}{1000\mathrm{s}}\right)^{\gamma}.$$

Here $P(x,t)$ is the rate in electrons per second that the pixel records t seconds following receiving a total number of electrons x. The parameters $A, x_0, \delta x, \alpha, \gamma$ may vary from pixel to pixel, though are presently fixed to global constants. This equation for the rate only applies to pixels which were illuminated more than to fill more than their half-well. We follow GalSim and linearly increase the persistence from 0 to the half-well value for illuminations between 0 and half-well.

This persistence rate is sampled with a Poisson distribution and added to each pixel read-by-read and incorporated into the resultants in the L1 images.

Persistence-affected pixels are expected to be rare, and are tracked sparsely via a list of the indices of affected pixels, the amount of the illumination, and the times of their illumination. Pixels are dropped from persistence tracking when their persistence rate is less than one electron per 100 seconds. If the same pixel is receives large fluxes multiple times, these are treated as two independent events and the resulting persistence flux is handled by summing the persistence rates given above over each event.

1.5.4 Cosmic rays

Cosmic rays are added to the simulation read-by-read. The cosmic ray parameters follow Wu et al. (2023). The locations of cosmic rays are chosen at random to sample the focal plane uniformly. Lengths are chosen according to a power law distribution $p(l)$
$siml^{-4.33}$, with lengths between 10 and 10,000 microns. Charge deposition rates per micron are selected from a Moyal distribution located at 120 electrons per micron with a width of 50 electrons per micron. An idealized charge is computed for each pixel in a read according to the product of the deposition rate per micron and the length of the cosmic ray's path within that pixel. This idealized charge is Poisson sampled and added to the relevant pixels in a read.

1.5.5 romanisim.l1 Module

Convert images into L1 images, with ramps.

We imagine starting with an image that gives the total number of counts from all Poisson processes (at least: sky, sources, dark current). We then need to redistribute these counts over the resultants of an L1 image.

The easiest thing to do, and probably where I should start, is to sample the image read-by-read with a binomial draw from the total counts weighted by the chance that each count landed in this particular time window. Then gather those for each resultant and average, as done on the spacecraft.

It's tempting to go straight to making the appropriate resultants. Following Casertano (2022?), the variance in each resultant is:

$$V = \sigma_{read}^2/N + f\tau$$

where f is the count rate, N is the number of reads in the resultant, and τ is the 'variance-based resultant time'

$$\tau = 1/N^2 \sum_{reads} (2(N-k)-1)t_k$$

where the t_k is the time of the kth read in the resultant.

For uniformly spaced reads,

$$\tau = t_0 + d(N/3 + 1/6N - 1/2)\,,$$

where t_0 is the time of the first read in the resultant and d is the spacing of the reads.

So that gives the variance from Poisson counts in resultant. But how should we draw random numbers to get that variance and the right mean? I can separately control the mean and variance by scaling the Poisson distribution, but I'm not sure that's doing the right thing with the higher order moments.

It probably isn't that expensive to just work out all of the reads, individually, which will also allow more natural incorporation of cosmic rays down the road. So let's take that approach instead for the moment.

How do we want to specify an L1 image? An L1 image is defined by a total count image and a list of lists $t_{i,j}$, where $t_{i,j}$ is the time at which the jth read in the ith resultant is made. We demand $t_{i,j} > t_{k,l}$ whenever i > k or whenever i = k and j > l.

Things this doesn't allow neatly:

- jitter in telescope pointing: the rate image is the same for each read/resultant
- weird non-linear systematics in darks?

Some systematics need to be applied to the individual reads, rather than to the final image. Currently linearity, persistenc, and CRs are implemented at individual read time. I need to think about when in the chain things like IPC, etc., come in. But it still seems correct to first generate the total number of counts that an ideal detector would measure from sources, and then apply these effects read-by-read as we build up the resultants—i.e., I expect the current framework will be able to handle this without major issue.

This approach is not super fast. For a high latitude set of resultants, generating all of the random numbers to determine the apportionment takes 43 s on the machine I'm currently using; this will scale linearly with the number of reads. That's longer than the actual image production for the dummy scene I'm using (though only ~2x longer).

I don't have a good way to speed this up. Explicitly doing the Poisson noise on each read from a rate image (rather than the apportionment of an image that already has Poisson noise) is 2x slower—generating billions of random numbers just takes a little while.

Plausibly I could figure out how to draw numbers directly from what a resultant is rather than looking at each read individually. That would likely bring a ~10x speed-up. The read noise there is easy. The poisson noise is a sum of scaled Poisson variables:

$$\sum_{i=0,...,N-1} (N-i)c_i\,,$$

where c_i is a Poisson-distributed variable. The sum of Poisson-distributed variables is Poisson-distributed, but I wasn't immediately able to find anything about the sum of scaled Poisson-distributed variables. The result is clearly not Poisson-distributed, but maybe there's some special way to sample from that directly.

If we did sample from that directly, we'd still also need to get the sum of the counts in the reads comprising the resultant. So I think you'd need a separate draw for that, conditional on the number you got for the resultant. Or, reversing that, you might want to draw the total number of counts first, e.g., via the binomial distribution, and then you'd want to draw a number for what the average number of counts was among the reads comprising the resultant, conditional on the total number of counts. Then

$$\sum_{i=0,...,N-1} (N-i)c_i$$

is some kind of statistic of the multinomial distribution. That sounds a little more tractable?

$$c_i \sim \mathrm{multinomial}(\mathrm{total}, [1/N, ..., 1/N])$$

We want to draw from $\sum(N-i)c_i$. I think the probabilities are always $1/N$, with the possible small but important exception of 'skipped' or 'dropped' reads, in which case the first read would be more like $2/(N+1)$ and all the others $1/(N+1)$. If the probabilities were always 1/N, this looks vaguely like it could have a nice analytic solution. Otherwise, I don't immediately see a route forward. So I am not going to pursue this avenue further.

Functions

add_ipc(resultants[, ipc_kernel])	Add IPC to resultants.
add_read_noise_to_resultants(resultants, tij)	Adds read noise to resultants.
apportion_counts_to_resultants(counts, tij)	Apportion counts to resultants given read times.
make_asdf(resultants[, dq, filepath, ...])	Package and optionally write out an L1 frame.
make_l1(counts, read_pattern[, read_noise, ...])	Make an L1 image from a counts image.
read_pattern_to_tij(read_pattern)	Get the times of each read going into resultants for a read_pattern.
tij_to_pij(tij[, remaining])	Convert a set of times tij to corresponding probabilities for sampling.
validate_times(tij)	Verify that a set of times tij for a valid resultant.

add_ipc

romanisim.l1.**add_ipc**(*resultants*, *ipc_kernel=None*)

Add IPC to resultants.

Parameters

resultants
[np.ndarray[n_resultant, nx, ny]] resultants describing scene

Returns

np.ndarray[n_resultant, nx, ny]
resultants with IPC

add_read_noise_to_resultants

romanisim.l1.**add_read_noise_to_resultants**(*resultants*, *tij*, *read_noise=None*, *rng=None*, *seed=None*)

Adds read noise to resultants.

The resultants get Gaussian read noise with sigma = sigma_read/sqrt(N). This is not quite right. In reality read noise is added during each read. This is the same as adding to the resultants and dividing by sqrt(N) except for quantization; this additional subtlety is currently ignored.

Parameters

resultants
[np.ndarray[n_resultant, nx, ny] (float)] resultants array, giving each of n_resultant resultant images

tij
[list[list[float]]] list of list of readout times for each read entering a resultant

read_noise
[float or np.ndarray[nx, ny] (float)] read noise or read noise image for adding to resultants

rng
[galsim.BaseDeviate] Random number generator to use

seed
[int] Seed for populating RNG. Only used if rng is None.

Returns

np.ndarray[n_resultant, nx, ny] (float)
resultants with added read noise

apportion_counts_to_resultants

romanisim.l1.**apportion_counts_to_resultants**(*counts*, *tij*, *inv_linearity=None*, *crparam=None*, *persistence=None*, *tstart=None*, *rng=None*, *seed=None*)

Apportion counts to resultants given read times.

This finds a statistically appropriate assignment of counts to each read composing a resultant, and averages the reads together to make the resultants.

There's an alternative approach where you have a count rate image and need to do Poisson draws from it. That's easier, and isn't this function. On some systems I've used Poisson draws have been slower than binomial draws, so it's not clear that approach offers any advantages, either— though I've had mixed experience there.

We loop over the reads, each time sampling from the counts image according to the probability that a photon lands in that particular read. This is just np.random.binomial(number of counts left, p/p_left)

We then average the reads together to get a resultant.

We accumulate:

- a sum for the resultant, which is divided by the number of reads and returned in the resultants array
- a sum for the total number of photons accumulated so far, so we know where to start the next resultant
- the resultants so far

Parameters

counts
[np.ndarray[nx, ny] (int)] The number of counts in each pixel from sources in the final image This final image should be a ~conceptual image of the scene observed by an idealized instrument seeing only backgrounds and sources and observing until the end of the last read; no instrumental effects are included beyond PSF & distortion.

tij
[list[list[float]]] list of list of readout times for each read entering a resultant

inv_linearity
[romanisim.nonlinearity.NL or None] Object implementing inverse non-linearity correction

crparam
[dict] Dictionary of keywords sent to romanisim.cr.simulate_crs for simulating cosmic rays. If None, no CRs are added

persistence
[romanisim.persistence.Persistence or None] Persistence object describing persistence-affected photons, or None if persistence should not be simulated.

tstart
[astropy.time.Time] Time of exposure start. Used only if persistence is not None.

rng
[galsim.BaseDeviate] random number generator

seed
[int] seed to use for random number generator

Returns

resultants, dq
resultants
[np.ndarray[n_resultant, nx, ny]] array of n_resultant images giving each resultant

dq
[np.ndarray[n_resultant, nx, ny]] dq array marking CR hits in resultants

make_asdf

romanisim.l1.**make_asdf**(*resultants*, *dq=None*, *filepath=None*, *metadata=None*, *persistence=None*)

Package and optionally write out an L1 frame.

This routine packages an L1 data file with the appropriate Roman data model. It currently does not do anything with the necessary metadata, and leaves that information as filler values.

Parameters

resultants
[np.ndarray[n_resultant, nx, ny] (float)] resultants array, giving each of n_resultant resultant images

filepath
[str] if not None, path of asdf file to L1 image into

dq
[np.ndarray[n_resultant, nx, ny] (int)] dq array flagging saturated / CR hit pixels

Returns

roman_datamodels.datamodels.ScienceRawModel
L1 image

extras
[dict] dictionary of additionally tabulated quantities, potentially including DQ images and persistence information.

make_l1

romanisim.l1.**make_l1**(*counts*, *read_pattern*, *read_noise=None*, *rng=None*, *seed=None*, *gain=None*, *inv_linearity=None*, *crparam=None*, *persistence=None*, *tstart=None*, *saturation=None*)

Make an L1 image from a counts image.

This apportions the total counts among the different resultants and adds some instrumental effects (linearity, IPC, CRs, persistence, . . .).

Parameters

counts
[galsim.Image] total counts delivered to each pixel

read_pattern
[int or list[list]] MA table number or list of lists giving indices of reads entering each resultant.

read_noise
[np.ndarray[nx, ny] (float) or float] Read noise entering into each read

rng
[galsim.BaseDeviate] Random number generator to use

seed
[int] Seed for populating RNG. Only used if rng is None.

gain
[float or np.ndarray[float]] Gain (electrons / count) for converting counts to electrons

inv_linearity
[romanisim.nonlinearity.NL or None] Object describing the inverse non-linearity corrections.

crparam
[dict] Keyword arguments to romanisim.cr.simulate_crs. If None, no cosmic rays are simulated.

persistence
[romanisim.persistence.Persistence] Persistence instance describing persistence-affected pixels

tstart
[astropy.time.Time] time of exposure start

Returns

l1, dq
l1: np.ndarray[n_resultant, nx, ny]
resultants image array including systematic effects

dq: np.ndarray[n_resultant, nx, ny]
DQ array marking saturated pixels and cosmic rays

read_pattern_to_tij

romanisim.l1.**read_pattern_to_tij**(*read_pattern*)

Get the times of each read going into resultants for a read_pattern.

Parameters

read_pattern
[int or list[list]] If int, id of ma_table to use. Otherwise a list of lists giving the indices of the reads entering each resultant.

Returns

list[list[float]]
list of list of readout times for each read entering a resultant

tij_to_pij

romanisim.l1.**tij_to_pij**(*tij*, *remaining=False*)

Convert a set of times tij to corresponding probabilities for sampling.

The probabilities are those needed for sampling from a binomial distribution for each read. These are delta_t / sum(delta_t), the fraction of time in each read over the total time, when *remaining* is False. When remaining is true, we scale these probabilities not by the total time but by the remaining time, so that subsequent reads get subsequent reads get scaled up so that each pij is delta_t / time_remaining, and the last read always has pij = 1.

Parameters

tij
[list[list[float]]] list of list of readout times for each read entering a resultant

remaining
[bool] scale by remaining time rather than total time

Returns

list[list[float]]
list of list of probabilities for each read, corresponding to the chance that a photon not yet assigned to a read so far should be assigned to this read.

validate_times

romanisim.l1.**validate_times**(*tij*)

Verify that a set of times tij for a valid resultant.

Parameters

tij
[list[list[float]]] a list of list of times at which each read in a resultant is performed

Returns

bool
True if the tij are ascending, otherwise False

1.5.6 romanisim.nonlinearity Module

Routines to handle non-linearity in simulating ramps.

The approach taken here is straightforward. The detector is accumulating photons, but the capacitance of the pixel varies with flux level and so the mapping between accumulated photons and read-out digital numbers changes with flux level. The CRDS linearity and inverse-linearity reference files describe the mapping between linear DN and observed DN. This module implements that mapping. When simulating an image, the photons entering each pixel are simulated, and then before being "read out" into a buffer, are transformed with this mapping into observed counts. These are then averaged and emitted as resultants.

Functions

evaluate_nl_polynomial(counts, coeffs[, ...])	Correct the observed counts for non-linearity.
repair_coefficients(coeffs, dq)	Fix cases of zeros and NaNs in non-linearity coefficients.

evaluate_nl_polynomial

romanisim.nonlinearity.**evaluate_nl_polynomial**(*counts*, *coeffs*, *reversed=False*)

Correct the observed counts for non-linearity.

As photons arrive, they make it harder for the device to count future photons due to classical non-linearity. This function converts some observed counts to what would have been seen absent non-linearity given some non-linearity corrections described by polynomials with given coefficients.

Parameters

counts
[np.ndarray[nx, ny] (float)] Number of counts already in pixel

coeffs
[np.ndarray[ncoeff, nx, ny] (float)] Coefficients of the non-linearity correction polynomials

reversed
[bool] If True, the coefficients are in reversed order, which is the order that np.polyval wants them. One can maybe save a little time reversing them once ahead of time.

Returns

corrected
[np.ndarray[nx, ny] (float)] The corrected number of counts

repair_coefficients

romanisim.nonlinearity.**repair_coefficients**(*coeffs*, *dq*)

Fix cases of zeros and NaNs in non-linearity coefficients.

This function replaces suspicious-looking non-linearity coefficients with no-op coefficients from a non-linearity perspective; all coefficients are zero except for the linear term, which is set to 1.

This function doesn't try to make sure that the derivative of the correction is greater than 1, which we would expect for a non-linearity correction.

Parameters

coeffs
[np.ndarray[ncoeff, nx, ny] (float)] Nonlinearity coefficients, starting with the constant term and increasing in power.

dq
[np.ndarray[n_resultant, nx, ny]] Data Quality array

Returns

coeffs
[np.ndarray[ncoeff, nx, ny] (float)] "repaired" coefficients with NaNs and weird coefficients replaced with linear values with slopes of unity.

dq
[np.ndarray[n_resultant, nx, ny]] DQ array marking pixels with improper non-linearity coefficients

Classes

NL(coeffs[, dq, gain])	Keep track of non-linearity and inverse non-linearity coefficients.

NL

class romanisim.nonlinearity.**NL**(*coeffs*, *dq=None*, *gain=None*)

Bases: object

Keep track of non-linearity and inverse non-linearity coefficients.

Construct an NL class handling non-linearity correction.

Parameters

coeffs
[np.ndarray[ncoeff, nx, ny] (float)] Non-linearity coefficients from reference files.

dq
[np.ndarray[n_resultant, nx, ny]] Data Quality array

gain
[float or np.ndarray[float]] Gain (electrons / count) for converting counts to electrons

Methods Summary

apply(counts[, electrons, reversed])	Compute the correction of observed to true counts

Methods Documentation

apply(*counts*, *electrons=False*, *reversed=False*)

Compute the correction of observed to true counts

Parameters

counts
[np.ndarray[nx, ny] (float)] The observed counts

electrons
[bool] Set to True for 'counts' being in electrons, with coefficients designed for DN. Accrdingly, the gain needs to be removed and reapplied.

reversed
[bool] If True, the coefficients are in reversed order, which is the order that np.polyval wants them. One can maybe save a little time reversing them once ahead of time.

Returns

corrected
[np.ndarray[nx, ny] (float)]

The corrected counts.

Class Inheritance Diagram

1.5.7 romanisim.persistence Module

Persistence module.

This module implements a persistence simulation following Sanchez+2023.

Functions

fermi(x, dt, A, x0, dx, alpha, gamma)	The Fermi model for persistence: A * (x/x0)**alpha * (t/1000.)**(-gamma) / (exp(-(x-x0)/dx) + 1) For influence level below the half well, the persistence is linear in x.

fermi

romanisim.persistence.**fermi**(*x*, *dt*, *A*, *x0*, *dx*, *alpha*, *gamma*)

The Fermi model for persistence: A * (x/x0)**alpha * (t/1000.)**(-gamma) / (exp(-(x-x0)/dx) + 1) For influence level below the half well, the persistence is linear in x.

Parameters

x
[np.ndarray[float]] Fluence level (electrons)

dt
[np.ndarray[float]] Time since exposure (s)

A
[float] Amplitude parameter of persistence (electrons)

x0
[float] Pivot fluence (electrons)

dx
[float] dx parameter (electrons)

alpha
[float] Power law index scaling with fluence

gamma
[float] Power law index scaling with time

Returns

The persistence signal at the current time for the persistence-affected pixels described by persistence.

Classes

Persistence([x, t, index, A, x0, dx, alpha, ...])	Track persistence information.

Persistence

class romanisim.persistence.**Persistence**(*x=None*, *t=None*, *index=None*, *A=None*, *x0=None*, *dx=None*, *alpha=None*, *gamma=None*)

Bases: object

Track persistence information.

There are two important sets of things to keep track of with persistence:

- how pixels respond to persistence
- what pixels have experienced large fluxes in the past and may be affected by persistence.

This class tracks both of those quantities. The first category is expected to be largely constant with time and basically only a function of the specific device doing the imaging. The second category changes in each exposure as new bright stars are observed.

Construct a new Persistence instance.

Parameters

x
[np.ndarray[float]] Fluence level (electrons)

t
[np.ndarray[float]] Time since exposure (s)

index
[np.ndarray[integer]] Indices of persistence-affected pixels (1D into raveled array)

A
[float] Amplitude parameter of persistence (electrons)

x0
[float] Pivot fluence (electrons)

dx
[float] dx parameter (electrons)

alpha
[float] Power law index scaling with fluence

gamma
[float] Power law index scaling with time

Methods Summary

add_to_read(image, tnow[, rng, seed])	Add persistence signature to image.
current(tnow)	Evaluate current in electron / s from past persistence artifacts at time tnow.
from_dict(d)	Convert a dictionary to a Persistence object.
read(filename)	Read a persistence dictionary from a simulated image.
to_dict()	Convert this persistence object to a dictionary.
update(image, tnow)	Update stored fluence values of events worth tracking for future persistence.
write(filename)	Write a persistence dictionary from a simulated image.

Methods Documentation

add_to_read(*image*, *tnow*, *rng=None*, *seed=50*)

Add persistence signature to image.

Parameters

image
[np.ndarray[float], shape: (npix_x, npix_y)] Image to which to add persistence (electrons)

tnow
[float] Current time (MJD)

rng
[np.random.Generator] Random number generator

seed
[int] Seed to use if instantiating new random number generator.

current(*tnow*)

Evaluate current in electron / s from past persistence artifacts at time tnow.

Parameters

tnow
[float] Current time (MJD)

Returns

Current in electron / s in pixels due to past persistence events.

static from_dict(*d*)

Convert a dictionary to a Persistence object.

Parameters

d
[dict] The dictionary representing the persistence object.

Returns

Persistence object represented by d.

`static read`(*filename*)

Read a persistence dictionary from a simulated image.

Parameters

filename
[str] The file name to read

Returns

Persistence object stored in filename.

`to_dict`()

Convert this persistence object to a dictionary.

Returns

dictionary representing persistence object.

`update`(*image*, *tnow*)

Update stored fluence values of events worth tracking for future persistence.

New persistence-affected pixels are added and old ones removed according to whether the predicted persistence rate is larger than parameters.persistence['ignorerate'].

Parameters

image
[np.ndarray[float]] Image of total electrons accumulated in exposure

tnow
[float] MJD of current observation

`write`(*filename*)

Write a persistence dictionary from a simulated image.

Parameters

filename
[str] The file name to read

Class Inheritance Diagram

1.5.8 romanisim.cr Module

Functions

create_sampler(pdf, x)	A function for performing inverse transform sampling.
moyal_distribution(x[, location, scale])	Return unnormalized Moyal distribution, which approximates a Landau distribution and is used to describe the energy loss probability distribution of a charged particle through a detector.
power_law_distribution(x[, slope])	Return unnormalized power-law distribution parameterized by a log-log slope, used to describe the cosmic ray path lengths.
sample_cr_params(N_samples[, N_i, N_j, ...])	Generates cosmic ray parameters randomly sampled from distribution.
simulate_crs(image, time[, flux, area, ...])	Adds CRs to an existing image.
traverse(trail_start, trail_end[, N_i, N_j, eps])	Given a starting and ending pixel, returns a list of pixel coordinates (ii, jj) and their traversed path lengths.

create_sampler

romanisim.cr.**create_sampler**(*pdf*, *x*)

A function for performing inverse transform sampling.

Parameters

pdf
[callable] A function or empirical set of tabulated values which can be used to call or evaluate *x*.

x
[1-d array of floats] A grid of values where the pdf should be evaluated.

Returns

inverse_cdf
[callable] Callable that gives the cumulative distribution function which allows sampling from the *pdf* distribution within the bounds described by the grid *x*.

moyal_distribution

romanisim.cr.**moyal_distribution**(*x*, *location=120*, *scale=50*)

Return unnormalized Moyal distribution, which approximates a Landau distribution and is used to describe the energy loss probability distribution of a charged particle through a detector.

Parameters

x
[1-d array] An array of dE/dx values (units: eV/micron) that forms the grid on which the Moyal distribution will be evaluated.

location
[float] The peak location of the distribution, units of eV / micron.

scale
[float] A width parameter for the distribution, units of eV / micron.

Returns
——-
moyal
[1-d array of floats] Moyal distribution (pdf) evaluated on *x* grid of points.

power_law_distribution

romanisim.cr.**power_law_distribution**(*x*, *slope=-4.33*)

Return unnormalized power-law distribution parameterized by a log-log slope, used to describe the cosmic ray path lengths.

Parameters

x
[1-d array of floats] An array of cosmic ray path lengths (units: micron).

slope
[float] The log-log slope of the distribution, default based on Miles et al. (2021).

Returns

power_law
[1-d array of floats] Power-law distribution (pdf) evaluated on *x* grid of points.

sample_cr_params

romanisim.cr.**sample_cr_params**(*N_samples*, *N_i=4096*, *N_j=4096*, *min_dEdx=None*, *max_dEdx=None*, *min_cr_len=None*, *max_cr_len=None*, *grid_size=None*, *rng=None*, *seed=48*)

Generates cosmic ray parameters randomly sampled from distribution.

Parameters

N_samples
[int] Number of CRs to generate.

N_i
[int] Number of pixels along i-axis of detector

N_j
[int] Number of pixels along j-axis of detector

min_dEdx
[float] Minimum value of CR energy loss (dE/dx), units of eV / micron.

max_dEdx
[float] Maximum value of CR energy loss (dE/dx), units of eV / micron.

min_cr_len
[float] Minimum length of cosmic ray trail, units of micron.

max_cr_len
[float] Maximum length of cosmic ray trail, units of micron.

grid_size
[int] Number of points on the cosmic ray length and energy loss grids. Increasing this parameter increases the level of sampling for the distributions.

rng
[np.random.Generator] Random number generator to use

seed
[int] seed to use for random number generator

Returns

cr_x
[float, between 0 and N_x-1] x pixel coordinate of cosmic ray, units of pixels.

cr_y
[float between 0 and N_y-1] y pixel coordinate of cosmic ray, units of pixels.

cr_phi
[float between 0 and 2pi] Direction of cosmic ray, units of radians.

cr_length
[float] Cosmic ray length, units of micron.

cr_dEdx
[float] Cosmic ray energy loss, units of eV / micron.

simulate_crs

romanisim.cr.**simulate_crs**(*image*, *time*, *flux=None*, *area=None*, *conversion_factor=None*, *pixel_size=None*, *pixel_depth=None*, *rng=None*, *seed=47*)

Adds CRs to an existing image.

Parameters

image
[2-d array of floats] The detector image with values in units of electrons.

time
[float] The exposure time, units of s.

flux
[float] Cosmic ray flux, units of cm^-2 s^-1. Default value of 8 is equal to the value assumed by the JWST ETC.

area
[float] The area of the WFI detector, units of cm^2.

conversion_factor
[float] The convert from eV to electrons, assumed to be the bandgap energy, in units of eV / electrons.

pixel_size
[float] The size of an individual pixel in the detector, units of micron.

pixel_depth
[float] The depth of an individual pixel in the detector, units of micron.

rng
[np.random.Generator] Random number generator to use

seed
[int] seed to use for random number generator

Returns

image
[2-d array of floats] The detector image, in units of electrons, updated to include all of the generated cosmic ray hits.

traverse

romanisim.cr.**traverse**(*trail_start*, *trail_end*, *N_i=4096*, *N_j=4096*, *eps=1e-10*)

Given a starting and ending pixel, returns a list of pixel coordinates (ii, jj) and their traversed path lengths. Note that the centers of pixels are treated as integers, while the borders are treated as half-integers.

Parameters

trail_start
[(float, float)] The starting coordinates in (i, j) of the cosmic ray trail, in units of pix.

trail_end
[(float, float)] The ending coordinates in (i, j) of the cosmic ray trail, in units of pix.

N_i
[int] Number of pixels along i-axis of detector

N_j
[int] Number of pixels along j-axis of detector

eps
[float] Tiny value used for stable numerical rounding.

Returns

ii
[np.ndarray[int]] i-axis positions of traversed trail, in units of pix.

jj
[np.ndarray[int]] j-axis positions of traversed trail, in units of pix.

lengths
[np.ndarray[float]] Chord lengths for each traversed pixel, in units of pix.

1.6 L2 images

L2 images are constructed from *L1* images, which are in turn constructed from idealized *count* images. This means that even when constructing L2 images, one must go through the process of simulating how counts get apportioned among the various reads. It is challenging to realistically model the statistics in the noise of L2 images without going through this process.

L2 images are constructed by doing "ramp fitting" on the level 1 images. Each pixel of a level 1 image is a series of "resultants", giving the measured value of that pixel averaged over a series on non-destructive reads as the exposure is being observed. A simple model for a pixel is that its flux as a function of time is simply two numbers: a pedestal and a linear ramp representing the rate at which photons are detected by the pixel. Ramp fitting turns these one-dimensional series of resultants into a "slope" image that is of interest astronomically. Due to details of the H4RG detectors, the pedestals of the ramp vary widely from exposure to exposure, and so current fitting completely throws away as non-astronomical any information in the ramp that is sensitive to the pedestal.

The package contains two algorithms for ramp fitting. The first uses "optimal" weighting, considering the full covariance matrix between each of the resultants stemming from read & Poisson noise from the sources. The covariance is inverted and combined with the design matrix in the usual least-squares approach to solve for the optimal slope and pedestal measurements for each pixel. This approach is naively expensive, but because the covariance matrices for each pixel for a one-dimensional family depending only on the ratio of the flux in the pixel to the read variance, the relevant matrices can be precomputed. These are then interpolated between for each pixel and summed over to get the parameters and variances for each pixel.

This approach does not handle cosmic rays or saturated pixels well, though for modest sized sets of resultants introducing an additional series of fits for the roughly $2n_{\rm resultant}$ sub-ramps would be straightforward. That approach would also naturally handle saturated ramps. Even explicitly computing every possible $n_{\rm resultant}(n_{\rm resultant}-1)/2$ subramp would likely still be quite inexpensive for modestly sized ramps.

The second approach follows Casertano+2022. In this approach, a diagonal set of weights is used in place of the full covariance matrix. The choice of weights depend on the particular pattern of reads assigned to each resultant and the amount of flux in the ramp, allowing them to interpolate from simply differencing the first and last resultants when the flux is very large to weighting the resultants by the number of reads when the flux is zero. This approach more efficiently handles dealing with ramps that have been split by cosmic rays, and obtaining uncertainties within a few percent of the "optimal" weighting approach. For these cases, we report final ramp slopes and variances derived from the inverse variance weighted subramp slopes and variances, using the read-noise derived variances.

This is a fairly faithful representation of how level two image construction works, so there are not many additional effects to add here. But mentioning some limitations:

- We have a simplistic saturation treatment, just clipping resultants that exceed the saturation level to the saturation level and throwing a flag.

1.6.1 romanisim.ramp Module

Ramp fitting routines.

The simulator need not actually fit any ramps, but we would like to do a good job simulating the noise induced by ramp fitting. That requires computing the covariance matrix coming out of ramp fitting. But that's actually a big part of the work of ramp fitting.

There are a few different proposed ramp fitting algorithms, differing in their weights. The final derived covariances are all somewhat similarly difficult to compute, however, since we ultimately end up needing to compute

$$(A^T C^{-1} A)^{-1}$$

for the "optimal" case, or

$$(A^T W^{-1} A)^{-1} A^T W^{-1} C W^{-1} A (A^T W^{-1} A)^{-1}$$

for some alternative weighting.

We start trying the "optimal" case below.

For the "optimal" case, a challenge is that we don't want to compute C^{-1} for every pixel individually. Fortunately, we only need $(A^T C^{-1} A)^{-1}$ (which is only a 2x2 matrix) for variances, and only $(A^T C^{-1} A)^{-1} A^T C^{-1}$ for ramp fitting, which is 2xn. Both of these matrices are effectively single parameter families, depending after rescaling by the read noise only on the ratio of the read noise and flux.

So the routines in these packages construct these different matrices, store them, and interpolate between them for different different fluxes and ratios.

Functions

construct_covar(read_noise, flux, read_pattern)	Constructs covariance matrix for first finite differences of unevenly sampled resultants.
construct_ki_and_variances(atcinva, atcinv, ...)	Construct the k_i weights and variances for ramp fitting.
construct_ramp_fitting_matrices(covar, ...)	Construct $A^TC^{-1}A$ and A^TC^{-1}, the matrices needed to fit ramps from resultants.
fit_ramps_casertano(resultants, dq, ...)	Fit ramps following Casertano+2022, including averaging partial ramps.
fit_ramps_casertano_no_dq(resultants, ...)	Fit ramps following Casertano+2022, only using full ramps.
ki_and_variance_grid(read_pattern, ...)	Construct a grid of k and covariances for the values of flux_on_readvar.
read_pattern_to_tau(read_pattern)	Construct the tau for each resultant from a read_pattern.
read_pattern_to_tbar(read_pattern)	Construct the mean times for each resultant from a read_pattern.
resultants_to_differences(resultants)	Convert resultants to their finite differences.
simulate_many_ramps([ntrial, flux, ...])	Simulate many ramps with a particular flux, read noise, and read_pattern.

construct_covar

romanisim.ramp.**construct_covar**(*read_noise*, *flux*, *read_pattern*)

Constructs covariance matrix for first finite differences of unevenly sampled resultants.

Parameters

read_noise
[float] The read noise (electrons)

flux
[float] The electrons per second

read_pattern
[list[list]] List of lists specifying the indices of the reads entering each resultant.

Returns

np.ndarray[n_resultant, n_resultant] (float)
covariance matrix of first finite differences of unevenly sampled resultants.

construct_ki_and_variances

romanisim.ramp.**construct_ki_and_variances**(*atcinva*, *atcinv*, *covars*)

Construct the k_i weights and variances for ramp fitting.

Following Casertano (2022), the ramp fit resultants are k.dot(differences), where $k = (A^TC^{-1}A)^{-1}A^TC^{-1}$, and differences is the result of resultants_to_differences(resultants). Meanwhile the variances are kCk^T. This function computes these k and variances.

Parameters

atcinva
[np.ndarray[2, 2] (float)] $A^TC^{-1}A$ from construct_ramp_fitting_matrices

atcinv
[np.ndarray[2, n_resultant] (float)] $A^T C^{-1}$ from construct_ramp_fitting_matrices

covars
[list[np.ndarray[n_resultant, n_resultant]]] covariance matrices to contract against k to compute variances

Returns

k
[np.ndarray[2, n_resultant]] $k = (A^T C^{-1} A)^{-} 1 A^T C^{-1}$ from Casertano (2022)

variances
[list[np.ndarray[2, 2]] (float)] $kC_i k^T$ for different covariance matrices C_i supplied in covars

construct_ramp_fitting_matrices

romanisim.ramp.**construct_ramp_fitting_matrices**(*covar*, *read_pattern*)

Construct $A^T C^{-1} A$ and $A^T C^{-1}$, the matrices needed to fit ramps from resultants.

The matrices constructed are those needed for applying to differences of resultants; e.g., the results of resultants_to_differences.

Parameters

covar
[np.ndarray[n_resultant, n_resultant] (float)] covariance of differences of resultants

read_pattern
[list[list]] List of lists specifying the reads entering each resultant

Returns

atcinva, atcinv
[np.ndarray[2, 2], np.ndarray[2, n_resultant] (float)] $A^T C^{-1} A$ and $A^T C^{-1}$, so that pedestal, flux = np.linalg.inv(atcinva).dot(atcinva.dot(differences))

fit_ramps_casertano

romanisim.ramp.**fit_ramps_casertano**(*resultants*, *dq*, *read_noise*, *read_pattern*)

Fit ramps following Casertano+2022, including averaging partial ramps.

Ramps are broken where dq != 0, and fits are performed on each sub-ramp. Resultants containing multiple ramps have their ramp fits averaged using inverse variance weights based on the variance in the individual slope fits due to read noise.

Parameters

resultants
[np.ndarry[nresultants, ...]] the resultants in electrons

dq
[np.ndarry[nresultants, ...]] the dq array. dq != 0 implies bad pixel / CR.

read noise: float
the read noise in electrons

read_pattern
[list[list[int]]] list of lists giving indices of reads entering each resultant

Returns

par
[np.ndarray[..., 2] (float)] the best fit pedestal and slope for each pixel

var
[np.ndarray[..., 3, 2, 2] (float)] the covariance matrix of par, for each of three noise terms: the read noise, Poisson source noise, and total noise.

fit_ramps_casertano_no_dq

romanisim.ramp.**fit_ramps_casertano_no_dq**(*resultants*, *read_noise*, *read_pattern*)

Fit ramps following Casertano+2022, only using full ramps.

This is a simpler implementation of fit_ramps_casertano, which doesn't address the case of partial ramps broken by CRs. This case is easier and can be done reasonably efficiently in pure python; results can be compared with fit_ramps_casertano in for the case of unbroken ramps.

Parameters

resultants
[np.ndarry[nresultants, npixel]] the resultants in electrons

read noise: float
the read noise in electrons

read_pattern
[list[list[int]]] list of lists giving indices of reads entering each resultant

Returns

par
[np.ndarray[nx, ny, 2] (float)] the best fit pedestal and slope for each pixel

var
[np.ndarray[nx, ny, 3, 2, 2] (float)] the covariance matrix of par, for each of three noise terms: the read noise, Poisson source noise, and total noise.

ki_and_variance_grid

romanisim.ramp.**ki_and_variance_grid**(*read_pattern*, *flux_on_readvar_pts*)

Construct a grid of k and covariances for the values of flux_on_readvar.

The k and corresponding covariances needed to do ramp fitting form essentially a one dimensional family in the flux in the ramp divided by the square of the read noise. This function constructs these quantities for a large number of different flux / read_noise^2 to be used in interpolation.

Parameters

read_pattern
[list[list] (int)] a list of lists of the indices of the reads entering each resultant

flux_on_readvar_pts
[array_like (float)] values of flux / read_noise**2 for which k and variances are desired.

Returns

kigrid
[np.ndarray[len(flux_on_readvar_pts), 2, n_resultants] (float)] k for each value of flux_on_readvar_pts

vargrid
[np.ndarray[len(flux_on_readvar_pts), n_covar, 2, 2] (float)] covariance of pedestal and slope corresponding to each value of flux_on_readvar_pts. n_covar = 3, for the contributions from read_noise, Poisson noise, and the sum.

read_pattern_to_tau

romanisim.ramp.**read_pattern_to_tau**(*read_pattern*)

Construct the tau for each resultant from a read_pattern.

$$\tau = \bar{t} - (n-1)(n+1)\delta t/6n$$

following Casertano (2022).

Parameters

read_pattern
[list[list]] List of lists specifying the indices of the reads entering each resultant.

Returns

τ
A time scale appropriate for computing variances.

read_pattern_to_tbar

romanisim.ramp.**read_pattern_to_tbar**(*read_pattern*)

Construct the mean times for each resultant from a read_pattern.

Parameters

read_pattern
[list[list]] List of lists specifying the indices of the reads entering each resultant.

Returns

tbar
[np.ndarray[n_resultant] (float)] The mean time of the reads of each resultant.

resultants_to_differences

romanisim.ramp.**resultants_to_differences**(*resultants*)

Convert resultants to their finite differences.

This is essentially np.diff(...), but retains the first resultant. The resulting structure has tri-diagonal covariance, which can be a little useful.

Parameters

resultants
[np.ndarray[n_resultant, nx, ny] (float)] The resultants

Returns

differences
[np.ndarray[n_resultant, nx, ny] (float)] Differences of resultants

simulate_many_ramps

romanisim.ramp.**simulate_many_ramps**(*ntrial=100*, *flux=100*, *readnoise=5*, *read_pattern=None*)

Simulate many ramps with a particular flux, read noise, and read_pattern.

To test ramp fitting, it's useful to be able to simulate a large number of ramps that are identical up to noise. This function does that.

Parameters

ntrial
[int] number of ramps to simulate

flux
[float] flux in electrons / s

read_noise
[float] read noise in electrons

read_pattern
[list[list] (int)] list of lists giving indices of reads entering each resultant

Returns

read_pattern
[list[list] (int)] read_pattern used

flux
[float] flux used

readnoise
[float] read noise used

resultants
[np.ndarray[n_resultant, ntrial] (float)] simulated resultants

Classes

RampFitInterpolator(read_pattern[, ...])	Ramp fitting tool aiding efficient fitting of large number of ramps.

RampFitInterpolator

class romanisim.ramp.**RampFitInterpolator**(*read_pattern*, *flux_on_readvar_pts=None*)

Bases: object

Ramp fitting tool aiding efficient fitting of large number of ramps.

The basic idea is that for a given image, ignoring cosmic rays or saturated pixels, the ramp fitting parameters are just a linear combination of the resultants. The weights of this linear combination are a single parameter family in the flux in the ramp divided by the read variance. So rather than explicitly calculating those weights for each pixel, we can up front calculate them overa grid in the flux over the read variance, and interpolate off that grid for each point. That can all be done in a vectorized way, allowing one to avoid doing something like a matrix inverse for each of a 16 million pixels.

The tool pre-calculates the grid and interpolators it needs at initialization, and then uses the results of that calculation when invoked to get the weights k or variances. The expectation is that most users just initialize and then call the fit_ramps method.

Construct a RampFitInterpolator for a read_pattern and a grid of flux/read_noise**2.

Parameters

read_pattern
[list[list] (int)] list of lists of indices of reads entering each resultant

flux_on_readvar_pts
[np.ndarray (float)] flux / read_noise**2 points at which to compute ramp fitting matrices. if None, a default grid will be used that should cover all reasonable values, from the read variance being 100k larger to 100k smaller than the electrons per second.

Methods Summary

fit_ramps(resultants, read_noise[, fluxest])	Fit ramps for a set of resultants and their read noise.
ki(flux, read_noise)	Compute k, the weights for the linear combination of resultant differences for optimal measurement of ramp pedestal and slope.
variances(flux, read_noise)	Compute the variances of ramp fit parameters.

Methods Documentation

fit_ramps(*resultants*, *read_noise*, *fluxest=None*)

Fit ramps for a set of resultants and their read noise.

Does not handle partial ramps (i.e., broken due to CRs).

Parameters

resultants
[np.ndarray[n_resultants, nx, ny] (numeric)] Resultants to fit

read_noise
[float or array_like like resultants] read noise in array

fluxest
[float or array_like like resultants] Initial estimate of flux in each ramp, in electrons per second. If None, estimated from the median flux differences between resultants.

Returns

par
[np.ndarray[nx, ny, 2] (float)] the best fit pedestal and slope for each pixel

var
[np.ndarray[nx, ny, 3, 2, 2] (float)] the covariance matrix of par, for each of three noise terms: the read noise, Poisson source noise, and total noise.

ki(*flux*, *read_noise*)

Compute k, the weights for the linear combination of resultant differences for optimal measurement of ramp pedestal and slope.

Parameters

flux
[array_like (float)] Estimate of electrons per second in ramp

read_noise
[array_like (float)] read_noise in ramp. Must be broadcastable with flux.

Returns

ki
[array_like[..., 2, n_resultant] (float)] k, weights of differences in linear combination of ramp pixels

`variances`(*flux*, *read_noise*)

Compute the variances of ramp fit parameters.

Parameters

flux
[array_like (float)] Estimate of electrons per second in ramp

read_noise
[array_like (float)] read_noise in ramp. Must be broadcastable with flux.

Returns

variances
[array_like[..., 3, 2, 2] (float)] covariance of ramp fit parameters, for read noise, poisson noise, and the total noise

Class Inheritance Diagram

1.7 Reference files

romanisim uses reference files from CRDS in order to simulate realistic images. The following kinds of reference files are used:

- read noise
- dark current
- flat field
- gain
- distortion map

The usage of these is mostly straightforward, but we provide here a few notes.

1.7.1 Read Noise

The random noise on reading a sample contributing to a ramp in an L1 image is scaled by the read noise reference file.

1.7.2 Dark Current

CRDS provides dark current images for each possible MA table, including the averaging of the dark current into resultants. This simplifies subtraction from L1 images and allows effects beyond a simple Poisson sampling of dark current electrons in each read. But it's unwieldy for a simulator because any effects beyond simple Poisson sampling of dark current electrons are not presently defined well enough to allow simulation. So the simulator simply takes the last resultant in the dark current resultant image and scales it by the effective exposure time of that resultant to get a dark current rate. This rate then goes into the idealized "counts" image which is then apportioned into the reads making up the resultants of an L1 image.

1.7.3 Flat field

Implementation of the flat field requires a little care due to the desire to support galsim's "photon shooting" rendering mode. This mode does not create noise-free images but instead only simulates the number of photons that would be actually detected in a device. We want to start by simulating the number of photons each pixel would record for a flat field of 1, and then sample that down by a fraction corresponding to the actual QE of each pixel. That works fine supposing that the flat field is less than 1, but does not work for values of the flat field greater than 1. So we instead do the initial galsim simulations for a larger exposure time than necessary, scaled by the maximum value of the flat field, and then sample down by $\mathrm{flat}/\mathrm{maxflat}$. That's all well and good as long as there aren't any spurious very large values in the flat field. I haven't actually seen any such values yet and so haven't tried to address that case (e.g., by clipping them).

1.7.4 Gain

Photons from the idealized "counts" image are scaled down to ADU before quantization during L1 creation, and then converted back to electrons before ramp fitting when making L2 images.

1.7.5 Distortion map

World coordinate systems for WFI images are created by placing the telescope boresight at $\mathrm{V2} = \mathrm{V3} = 0$, and then applying the distortion maps from CRDS to convert from V2V3 to pixels.

1.8 Catalogs

The simulator takes catalogs describing objects in a scene and generates images of that scene. These catalogs have the following form:

```
   ra     dec   type    n    half_light_radius    pa      ba     F087
float64 float64 str3 float64      float64      float64 float64 float64
------- ------- ---- ------- ------------------ ------- ------- --------
  269.9    66.0  SER     1.6                0.6   165.6     0.9 1.80e-09
  270.1    66.0  SER     3.6                0.4    71.5     0.7 3.35e-09
  269.8    66.0  PSF    -1.0                0.0     0.0     1.0 2.97e-10
  269.9    66.0  SER     2.5                0.8   308.8     0.7 1.50e-09
```

(continues on next page)

(continued from previous page)

```
 269.8    65.9  SER     3.9              0.9   210.0     0.9 3.28e-10
 270.1    66.0  SER     4.0              1.1   225.1     1.0 1.61e-09
 269.9    65.9  SER     1.5              0.3   271.8     0.6 1.13e-09
 269.9    65.9  SER     2.9              2.3    27.6     1.0 3.28e-09
 269.9    66.0  SER     1.1              0.3     4.3     1.0 9.99e-10
```

The following fields must be specified for each source:

- ra: the right ascension of the source
- dec: the declination of the source
- type: PSF or SER; whether the source is a point source or Sersic galaxy
- n: the Sersic index of the source. This value is ignored if the source is a point source.
- half_light_radius: the half light radius of the source in arcseconds. This value is ignored if the source is a point source.
- pa: the position angle of the source, in degrees east of north. This value is ignored if the source is a point source.
- ba: the major-to-minor axis ratio. This value is ignored if the source is a point source.

Following these required fields is a series of columns giving the fluxes of the the sources in "maggies"; the AB magnitude of the source is given by $-2.5 * \log_{10}(\mathrm{flux})$. In order to simulate a scene in a given bandpass, a column with the name of that bandpass must be present giving the total fluxes of the sources. Many flux columns may be present, and other columns may also be present but will be ignored.

The simulator then renders these images in the scene and produces the simulated L1 or L2 images.

The simulator API includes a few simple tools to generate parametric distributions of stars and galaxies. The `make_stars` and `make_galaxies` routines make random catalogs of stars and galaxies. The number of stars and galaxies can be adjusted. Likewise, the power law index by which the sources' magnitudes are sampled can be adjusted, as can their limiting magnitudes. Galaxy Sersic parameters, half-light radii, and position angles are chosen at random, with a rough attempt to make brighter galaxies appropriately larger (i.e., conserving surface brightness). Stars can be chosen to be distributed with a King profile. This functionality is however very rudimentary and limited, and is better suited for toy problems than real scientific work. We expect scientific uses to be driven by custom-created catalogs rather than these simple routines.

1.8.1 romanisim.catalog Module

Catalog generation and reading routines.

This module provides basic routines to allow romanisim to render scenes based on catalogs of sources in those scenes.

Functions

make_dummy_catalog(coord[, radius, rng, ...])	Make a dummy catalog for testing purposes.
make_dummy_table_catalog(coord[, radius, ...])	Make a dummy table catalog.
make_galaxies(coord, n[, radius, index, ...])	Make a simple parametric catalog of galaxies.
make_stars(coord, n[, radius, index, ...])	Make a simple parametric catalog of stars.
read_catalog(filename, bandpasses)	Read a catalog into a list of CatalogObjects.
table_to_catalog(table, bandpasses)	Read a astropy Table into a list of CatalogObjects.

make_dummy_catalog

romanisim.catalog.**make_dummy_catalog**(*coord*, *radius=0.1*, *rng=None*, *seed=42*, *nobj=1000*, *chromatic=True*, *galaxy_sample_file_name=None*)

Make a dummy catalog for testing purposes.

Parameters

coord
[galsim.CelestialCoordinate] center around which to generate sources

radius
[float] radius (deg) within which to generate sources

rng
[Galsim.BaseDeviate] Random number generator to use

seed
[int] Seed for populating random number generator. Only used if rng is None.

nobj
[int] Number of objects to simulate.

chromatic
[bool] Use chromatic objects rather than gray objects. The PSF of chromatic objects depends on their SED, while for gray objects this dependence is neglected.

Returns

list[CatalogObject]
list of catalog objects to render

make_dummy_table_catalog

romanisim.catalog.**make_dummy_table_catalog**(*coord*, *radius=0.1*, *rng=None*, *nobj=1000*, *bandpasses=None*, *seed=None*)

Make a dummy table catalog.

Fluxes are assigned to bands at random. Locations are random within the spherical cap defined by coord and radius.

Parameters

coord
[astropy.coordinates.SkyCoord] Location around which to generate catalog

radius
[float] Radius in degrees of spherical cap in which to generate sources

rng
[galsim.BaseDeviate] Random number generator to use

nobj
[int] Number of objects to generate in spherical cap.

bandpasses
[list[str]] List of names of bandpasses in which to generate fluxes.

Returns

astropy.table.Table
Table including fields needed to generate a list of CatalogObject entries for rendering.

make_galaxies

romanisim.catalog.**make_galaxies**(*coord*, *n*, *radius=0.1*, *index=None*, *faintmag=26*, *hlr_at_faintmag=0.6*, *bandpasses=None*, *rng=None*, *seed=50*)

Make a simple parametric catalog of galaxies.

Parameters

coord
[astropy.coordinates.SkyCoord] Location around which to generate sources.

n
[int] number of sources to generate

radius
[float] radius in degrees of cap in which to uniformly generate sources

index
[int] power law index of magnitudes

faintmag
[float] faintest AB magnitude for which to generate sources Note this magnitude is in a "fiducial" band which is not observed. Actual requested bandpasses are equal to this fiducial band plus 1 mag of Gaussian noise.

hlr_at_faintmag
[float] typical half light radius at faintmag (arcsec)

bandpasses
[list[str]] list of names of bandpasses for which to generate fluxes.

rng
[galsim.BaseDeviate] random number generator to use

seed
[int] seed to use for random numbers, only used if rng is None

Returns

catalog
[astropy.Table] Table for use with table_to_catalog to generate catalog for simulation.

make_stars

romanisim.catalog.**make_stars**(*coord*, *n*, *radius=0.1*, *index=None*, *faintmag=26*, *truncation_radius=None*, *bandpasses=None*, *rng=None*, *seed=51*)

Make a simple parametric catalog of stars.

If truncation radius is None, this makes a uniform distribution. If the truncation_radius is not None, it makes a King distribution where the core radius is given by the radius and the truncation radius is given by truncation_radius.

Parameters

coord
[astropy.coordinates.SkyCoord] Location around which to generate sources.

n
[int] number of sources to generate

radius
[float] radius in degrees of cap in which to generate sources

index
[int] power law index of magnitudes; uniform density & standard candle implies 3/5.

faintmag
[float] faintest AB magnitude for which to generate sources Note this magnitude is in a "fiducial" band which is not observed. Actual requested bandpasses are equal to this fiducial band plus 1 mag of Gaussian noise.

truncation_radius
[float] truncation radius of cluster if not None; otherwise ignored.

bandpasses
[list[str]] list of names of bandpasses for which to generate fluxes.

rng
[galsim.BaseDeviate] random number generator to use

seed
[int] seed for random number generator to use, only used if rng is None

Returns

catalog
[astropy.Table] Table for use with table_to_catalog to generate catalog for simulation.

read_catalog

romanisim.catalog.**read_catalog**(*filename*, *bandpasses*)

Read a catalog into a list of CatalogObjects.

Catalog must be readable by astropy.table.Table.read(...) and contain columns enumerated in the docstring for table_to_catalog(...).

Parameters

filename
[str] filename of catalog to read

bandpasses
[list[str]] bandpasses for which fluxes are tabulated in the catalog

Returns

list[CatalogObject]
list of catalog objects in filename

table_to_catalog

romanisim.catalog.**table_to_catalog**(*table*, *bandpasses*)

Read a astropy Table into a list of CatalogObjects.

We want to read in a catalog and make a list of CatalogObjects. The table must have the following columns:

- ra : float, right ascension in degrees
- dec : float, declination in degrees
- type : str, 'PSF' or 'SER' for PSF or sersic profiles respectively
- n : float, sersic index
- half_light_radius : float, half light radius in arcsec
- pa : float, position angle of ellipse relative to north (on the sky) in degrees
- ba : float, ratio of semiminor axis b over semimajor axis a

Additionally there must be a column for each bandpass giving the flux in that bandbass.

Parameters

table
[astropy.table.Table] astropy Table containing ra, dec, type, n, half_light_radius, pa, ba and fluxes in different bandpasses

bandpasses
[list[str]] list of names of bandpasses. These bandpasses must have columns of the corresponding names in the catalog, containing the objects' fluxes.

Returns

list[CatalogObject]
list of catalog objects for catalog

Classes

CatalogObject(sky_pos, profile, flux)	Simple class to hold galsim positions and profiles of objects.

CatalogObject

class romanisim.catalog.**CatalogObject**(*sky_pos: coord.CelestialCoord*, *profile: galsim.GSObject*, *flux: dict*)

Bases: object

Simple class to hold galsim positions and profiles of objects.

Flux element contains the total AB flux from the source; i.e., the -2.5*log10(flux[filter_name]) would be the AB magnitude of the source.

Class Inheritance Diagram

1.9 APT file support

The simulator possesses rudimentary support for simulating images from APT files. In order to simulate a scene, romanisim needs to know what's in the scene, as specified by a *catalog*. It also needs to know where the telescope is pointed, the roll angle of the telescope, the date of the observation, and the bandpass. Finally, it needs to know what the MultiAccum table of the observation is—roughly, how long the exposure is and how the reads of the detector should be averaged into resultants.

Much of this information is available in an APT file. A rudimentary APT file reader can pull out the right ascension and declinations of observations, as well as the filters requested. However, support for roll angles is not yet included. APT files do not include the dates of observation, so this likewise is not included. APT files naturally do not contain catalogs of sources in the field, so some provision must be made for adding this information.

This module is not yet fully baked.

1.9.1 romanisim.apt Module

Very simple APT reader.

Converts an APT file into a list of (ra, dec, angle, filter, date, exposure time) needed for generating observations. This is adequate for reading in a few of the example Roman APTs but only supports a tiny fraction of what an APT file seems able to do.

Functions

read_apt(filename)	Read an APT file, returning a list of observations.

read_apt

romanisim.apt.**read_apt**(*filename*)

Read an APT file, returning a list of observations.

Parameters

filename
[str] filename of the APT file to read in.

Returns

list[Observation]
list of Observations in the APT file

Classes

Observation(target, bandpass, exptime, date)	An observation of a target.
Target(name, number, coords)	A target for observation.

Observation

class romanisim.apt.**Observation**(*target:* Target, *bandpass: str*, *exptime: float*, *date: datetime*)

Bases: object

An observation of a target.

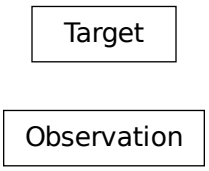
Target

class romanisim.apt.**Target**(*name: str*, *number: int*, *coords: astropy.coordinates.SkyCoord*)

Bases: object

A target for observation.

Class Inheritance Diagram



1.10 Bandpasses

The simulator can render scenes in a number of different bandpasses. The choice of bandpass affects the point spread function used, the sky backgrounds, the fluxes of sources, and the reference files requested.

At present, romanisim simply passes the choice of bandpass to other packages—to webbpsf for PSF modeling, to galsim.roman for sky background estimation, to CRDS for reference file selection, or to the catalog for the selection of appropriate fluxes. However, because catalog fluxes are specified in "maggies" (i.e., in linear fluxes on the AB scale), the simulator needs to know how to convert between a maggie and the number of photons Roman receives from a source. Accordingly, the simulator knows about the AB zero points of the Roman filters, as derived from https://roman.gsfc.nasa.gov/science/WFI_technical.html .

One technical note: it is unclear what aperture is used for the bandpasses provided by Goddard. The Roman PSF formally extends to infinity and some light is received by the detector but is so far from the center of the PSF that it is not useful for flux, position, or shape measurements. Often for the purposes of computing effective area curves only light landing within a fixed aperture is counted. Presently the simulator assumes that an infinite aperture is used. This can result in an approximately 10% different flux scale than more reasonable aperture selections.

1.10.1 romanisim.bandpass Module

Roman bandpass routines

The primary purpose of this module is to provide the number of counts per second expected for sources observed by Roman given a source with the nominal flat AB spectrum of 3631 Jy. The ultimate source of this information is https://roman.gsfc.nasa.gov/science/WFI_technical.html .

Functions

compute_abflux([effarea])	Compute the AB zero point fluxes for each filter.
compute_count_rate(flux, bandpass[, ...])	Compute the count rate in a given filter, for a specified SED.
get_abflux(bandpass)	Get the zero point flux for a particular bandpass.
read_gsfc_effarea([filename])	Read an effective area file from Roman.

compute_abflux

romanisim.bandpass.**compute_abflux**(*effarea=None*)

Compute the AB zero point fluxes for each filter.

How many photons would a zeroth magnitude AB star deposit in Roman's detectors in a second?

Parameters

effarea
[astropy.Table.table] Table from GSFC with effective areas for each filter.

Returns

dict[str]
[float] lookup table of zero point fluxes for each filter (photons / s)

compute_count_rate

romanisim.bandpass.**compute_count_rate**(*flux*, *bandpass*, *filename=None*, *effarea=None*, *wavedist=None*)

Compute the count rate in a given filter, for a specified SED.

How many photons would an object with SED given by flux deposit in Roman's detectors in a second?

Parameters

flux
[float or np.ndarray with shape matching wavedist.] Spectral flux density in units of ergs per second * hertz * cm^2

bandpass
[str] the name of the bandpass

filename
[str] filename to read in

effarea
[astropy.Table.table] Table from GSFC with effective areas for each filter.

wavedist
[numpy.ndarray] Array of wavelengths along which spectral flux densities are defined in microns

Returns

float
the total bandpass flux (photons / s)

get_abflux

romanisim.bandpass.**get_abflux**(*bandpass*)

Get the zero point flux for a particular bandpass.

This is a simple wrapper for compute_abflux, caching the results.

Parameters

bandpass
[str] the name of the bandpass

Returns

float
the zero point flux (photons / s)

read_gsfc_effarea

romanisim.bandpass.**read_gsfc_effarea**(*filename=None*)

Read an effective area file from Roman.

This just puts together the right invocation to get an Excel-converted CSV file into memory.

Parameters

filename
[str] filename to read in

Returns

astropy.table.Table
table with effective areas for different Roman bandpasses.

1.11 Point Spread Function Modeling

The simulator has two mechanisms for point source modeling. The first uses the galsim implementation of the Roman point spread function; for more information, see the galsim Roman documentation. The second uses the webbpsf package to make a model of the Roman PSF.

In the current implementation, the simulator uses a linearly varying, achromatic bandpass for each filter when using webbpsf. That is, the PSF does not vary depending on the spectrum of the source being rendered. However, it seems straightforward to implement either of these modes in the context of galsim, albeit at some computational expense.

When using the galsim PSF, galsim's "photon shooting" mode is used for efficient rendering of chromatic sources. When using webbpsf, FFTs are used to to do the convolution of the intrinsic source profile with the PSF and pixel grid of the instrument.

1.11.1 romanisim.psf Module

Roman PSF interface for galsim.

galsim.roman has an implementation of Roman's PSF based on the aperture and some estimates for the wavefront errors over the aperture as described by amplitudes of various Zernicke modes. This seems like a very good approach, but we want to add here a mode using the official PSFs coming out of webbpsf, which takes a very similar overall approach.

galsim's InterpolatedImage class makes this straightforward. Future work should consider the following:

- how do we want to deal with the dependence of the PSF on the source SED? It's possible I can just subclass ChromaticObject and implement evaluateAtWavelength, possibly also stealing the _shoot code from ChromaticOpticalPSF?

Functions

make_one_psf(sca, filter_name[, wcs, ...])	Make a PSF profile for Roman at a specific detector location.
make_psf(sca, filter_name[, wcs, webbpsf, ...])	Make a PSF profile for Roman.

make_one_psf

romanisim.psf.**make_one_psf**(*sca*, *filter_name*, *wcs=None*, *webbpsf=True*, *pix=None*, *chromatic=False*, *oversample=4*, ***kw*)

Make a PSF profile for Roman at a specific detector location.

Can construct both PSFs using galsim's built-in galsim.roman.roman_psfs routine, or can use webbpsf.

Parameters

sca
[int] SCA number

filter_name
[str] name of filter

wcs
[callable (optional)] function giving mapping from pixels to sky for use in computing local scale of image for webbpsf PSFs

pix
[tuple (float, float)] pixel location of PSF on focal plane

oversample
[int] oversampling with which to sample WebbPSF PSF

****kw**
[dict] Additional keywords passed to galsim.roman.getPSF or webbpsf.calc_psf, depending on whether webbpsf is set.

Returns

profile
[galsim.gsobject.GSObject] galsim profile object for convolution with source profiles when rendering scenes.

make_psf

romanisim.psf.**make_psf**(*sca*, *filter_name*, *wcs=None*, *webbpsf=True*, *pix=None*, *chromatic=False*, *variable=False*, ***kw*)

Make a PSF profile for Roman.

Optionally supports spatially variable PSFs via interpolation between the four corners.

Parameters

sca
[int] SCA number

filter_name
[str] name of filter

wcs
[callable (optional)] function giving mapping from pixels to sky for use in computing local scale of image for webbpsf PSFs

pix
[tuple (float, float)] pixel location of PSF on focal plane

variable
[bool] True if a variable PSF object is desired

****kw**
[dict] Additional keywords passed to make_one_psf

Returns

profile
[galsim.gsobject.GSObject] galsim profile object for convolution with source profiles when rendering scenes.

Classes

VariablePSF(corners, psf)	Spatially variable PSF wrapping GalSim profiles.

VariablePSF

class romanisim.psf.**VariablePSF**(*corners*, *psf*)

Bases: object

Spatially variable PSF wrapping GalSim profiles.

Linearly interpolates between four corner PSF profiles by summing weighted GalSim PSF profiles.

Methods Summary

at_position(x, y)	Instantiate a PSF profile at (x, y).

Methods Documentation

at_position(*x*, *y*)

Instantiate a PSF profile at (x, y).

Linearly interpolate between the four corners to obtain the PSF at this location.

Parameters

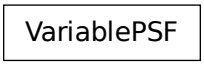
x
[float] x position

y
[float] y position

Returns

GalSim profile representing PSF at (x, y).

Class Inheritance Diagram



1.12 World Coordinate Systems & Distortion

The simulator has two options for modeling distortion and world coordinate systems. The first is to use the routines in the galsim.roman package; see GalSim's documentation for more information. The second is to use distortion reference files from the Calibration References Data System (CRDS).

The latter system works by grabbing reference distortion maps for the appropriate detector and filter combinations from the Roman CRDS server. These distortion maps are then wrapped into a galsim WCS object and fed to galsim's rendering pipeline.

1.12.1 romanisim.wcs Module

Roman WCS interface for galsim.

galsim.roman has an implementation of Roman's WCS based on some SIP coefficients for each SCA. This is presumably plenty good, but here we take the alternative approach of using the distortion functions provided in CRDS. These naturally are handled by the gWCS library, but galsim only naturally supports astropy WCSes via the ~legacy interface. So this module primarily makes the wrapper that interfaces gWCS and galsim.CelestialWCS together.

This presently gives rather different world coordinates given a specific telescope boresight. Partially this is not doing the same roll_ref determination that galsim.roman does, which could be fixed. But additionally the center of the SCA looks to be in a different place relative to the boresight for galsim.roman than for what I get from CRDS. This bears more investigation.

Functions

convert_wcs_to_gwcs(wcs)	Convert a GalSim WCS object into a GWCS object.
fill_in_parameters(parameters, coord[, ...])	Add WCS info to parameters dictionary.
get_wcs(image[, usecrds, distortion])	Get a WCS object for a given sca or set of CRDS parameters.
make_wcs(targ_pos, distortion[, roll_ref, ...])	Create a gWCS from a target position, a roll, and a distortion map.
wcs_from_fits_header(header)	Convert a FITS WCS to a GWCS.

convert_wcs_to_gwcs

romanisim.wcs.**convert_wcs_to_gwcs**(*wcs*)

Convert a GalSim WCS object into a GWCS object.

Parameters

wcs
[gwcs.wcs.WCS or wcs.GWCS] input WCS to convert

Returns

wcs.GWCS corresponding to wcs.

fill_in_parameters

romanisim.wcs.**fill_in_parameters**(*parameters*, *coord*, *pa_aper=0*, *boresight=True*)

Add WCS info to parameters dictionary.

Parameters

parameters
[dict] Metadata dictionary Dictionaries like pointing, aperture, and wcsinfo may be modified

coord
[astropy.coordinates.SkyCoord or galsim.CelestialCoord] world coordinates at V2 / V3 ref (boresight or center of WFI CCDs)

pa_aper
[float] position angle (North to YIdl) at the aperture V2Ref/V3Ref

boresight
[bool] whether coord is the telescope boresight (V2 = V3 = 0) or the center of the WFI CCD array

get_wcs

romanisim.wcs.**get_wcs**(*image*, *usecrds=True*, *distortion=None*)

Get a WCS object for a given sca or set of CRDS parameters.

Parameters

image
[roman_datamodels.datamodels.ImageModel or dict] Image model or dictionary containing CRDS parameters specifying appropriate reference distortion map to load.

usecrds
[bool] If True, use crds reference distortions rather than galsim.roman distortion model.

distortion
[astropy.modeling.core.CompoundModel] Coordinate distortion transformation parameters

Returns

galsim.CelestialWCS for an SCA

make_wcs

romanisim.wcs.**make_wcs**(*targ_pos*, *distortion*, *roll_ref=0*, *v2_ref=0*, *v3_ref=0*, *wrap_v2_at=180*, *wrap_lon_at=360*)

Create a gWCS from a target position, a roll, and a distortion map.

Parameters

targ_pos
[astropy.coordinates.SkyCoord] The celestial coordinates of the boresight or science aperture.

distortion
[callable] The distortion mapping pixel coordinates to V2/V3 coordinates for a detector.

roll_ref
[float] The angle of the V3 axis relative to north, increasing from north to east, at the boresight or science aperture. Note that the V3 axis is rotated by +60 degree to the +Y axis.

v2_ref
[float] The v2 coordinate (arcsec) corresponding to targ_pos

v3_ref
[float] The v3 coordinate (arcsec) corresponding to targ_pos

Returns

gwcs.wcs object representing WCS for observation

wcs_from_fits_header

romanisim.wcs.**wcs_from_fits_header**(*header*)

Convert a FITS WCS to a GWCS.

This function reads SIP coefficients from a FITS WCS and implements the corresponding gWCS WCS. It was copied from gwcs.tests.utils._gwcs_from_hst_fits_wcs.

Parameters

header
[astropy.io.fits.header.Header] FITS header

Returns

wcs
[gwcs.wcs.WCS] gwcs WCS corresponding to header

Classes

GWCS(gwcs[, origin])	This WCS uses gWCS to implent a galsim CelestialWCS.

GWCS

class romanisim.wcs.**GWCS**(*gwcs*, *origin=None*)

Bases: CelestialWCS

This WCS uses gWCS to implent a galsim CelestialWCS.

Based on galsim.fitswcs.AstropyWCS, edited to eliminate header functionality and to adopt the shared API supported by both gWCS and astropy.wcs.

Parameters

gwcs
[gwcs.WCS] The WCS object to wrap in a galsim CelestialWCS interface.

Attributes Summary

origin	The origin in image coordinates of the WCS function.
wcs	The underlying gwcs.WCS object.

Methods Summary

copy()

Attributes Documentation

origin

The origin in image coordinates of the WCS function.

wcs

The underlying gwcs.WCS object.

Methods Documentation

copy()

Class Inheritance Diagram

1.13 Parameters

The parameters module contains useful constants describing the Roman telescope. These include:

- the default read noise when CRDS is not used
- the number of border pixels
- the specification of the read indices going into resultants for a fiducial L1 image for a handful of MA tables
- the read time
- the default saturation limit
- the default IPC kernel

- the definition of the reference V2/V3 location in the focal plane to which to place the given ra/dec
- the default persistence parameters
- the default cosmic ray parameters

These values can be overridden by specifying a yaml config file on the command line to romanisim-make-image.

1.13.1 romanisim.parameters Module

Parameters class storing a few useful constants for Roman simulations.

1.14 Utilities

The simulator utility module consists of miscellaneous utility routines intended to be of broad use. Present examples include:

- turning astropy coordinates into galsim coordinates and vice-versa
- making an RGB image from image slices
- generating points at random in a region on the sky

1.14.1 romanisim.util Module

Miscellaneous utility routines.

Functions

add_more_metadata(metadata)	Fill out the metadata dictionary, modifying it in place.
celestialcoord(sky)	Turn a SkyCoord into a CelestialCoord.
king_profile(r, rc, rt)	Compute the King (1962) profile.
random_points_at_radii(coord, radii[, rng])	Choose locations at random at given radii from coord.
random_points_in_cap(coord, radius, nobj[, rng])	Choose locations at random within radius of coord.
random_points_in_king(coord, rc, rt, nobj[, rng])	Sample points from a King distribution
sample_king_distances(rc, rt, npts[, rng])	Sample distances from a King (1962) profile.
scalergb(rgb[, scales, lumrange])	Scales three flux images into a range of luminosity for displaying.
skycoord(celestial)	Turn a CelestialCoord into a SkyCoord.

add_more_metadata

romanisim.util.**add_more_metadata**(*metadata*)

Fill out the metadata dictionary, modifying it in place.

Parameters

metadata
[dict] CRDS-style dictionary containing keywords like roman.meta.exposure.start_time.

celestialcoord

romanisim.util.**celestialcoord**(*sky*)

Turn a SkyCoord into a CelestialCoord.

Parameters

sky
[astropy.coordinates.SkyCoord] astropy.coordinates.SkyCoord to transform into an galsim.CelestialCoord

Returns

galsim.CelestialCoord
CelestialCoord corresponding to skycoord

king_profile

romanisim.util.**king_profile**(*r*, *rc*, *rt*)

Compute the King (1962) profile.

Parameters

r
[np.ndarray[float]] distances at which to evaluate the King profile

rc
[float] core radius

rt
[float] truncation radius

Returns

2D number density of stars at r.

random_points_at_radii

romanisim.util.**random_points_at_radii**(*coord*, *radii*, *rng=None*)

Choose locations at random at given radii from coord.

Parameters

coord
[astropy.coordinates.SkyCoord] location around which to generate points

distances
[astropy.Quantity[float]] angular distances points should lie from center

rng
[galsim.UniformDeviate] random number generator to use

Returns

astropy.coordinates.SkyCoord

random_points_in_cap

romanisim.util.**random_points_in_cap**(*coord*, *radius*, *nobj*, *rng=None*)

Choose locations at random within radius of coord.

Parameters

coord
[astropy.coordinates.SkyCoord] location around which to generate points

radius
[float] radius in deg of region in which to generate points

nobj
[int] number of objects to generate

rng
[galsim.UniformDeviate] random number generator to use

Returns

astropy.coordinates.SkyCoord

random_points_in_king

romanisim.util.**random_points_in_king**(*coord*, *rc*, *rt*, *nobj*, *rng=None*)

Sample points from a King distribution

Parameters

coord
[astropy.coordinates.SkyCoord] location around which to generate points

rc
[float] core radius in deg

rt
[float] truncation radius in deg

nobj
[int] number of objects to generate

rng
[galsim.UniformDeviate] random number generator to use

Returns

astropy.coordinates.SkyCoord

sample_king_distances

romanisim.util.**sample_king_distances**(*rc*, *rt*, *npts*, *rng=None*)

Sample distances from a King (1962) profile.

Parameters

rc
[float] core radius

rt
[float] truncation radius

npts
[int] number of points to generate

rng
[galsim.BaseDeviate] random number generator to use

Returns

r
[float] Distances distributed according to a King (1962) profile.

scalergb

romanisim.util.**scalergb**(*rgb*, *scales=None*, *lumrange=None*)

Scales three flux images into a range of luminosity for displaying.

Images are scaled into [0, 1].

This routine is intended to help with cases where you want to display some images and want the color scale to cover only a certain range, but saturated regions should retain their appropriate hue and not be compressed to white.

Parameters

rgb
[np.ndarray[npix, npix, 3]] the RGB images to scale

scales
[list[float] (must contain 3 floats)] rescale each image by this amount

lumrange
[list[float] (must contain 2 floats)] minimum and maximum luminosity

Returns

im
[np.ndarray[npix, npix, 3]] scaled RGB image suitable for displaying

skycoord

romanisim.util.**skycoord**(*celestial*)

Turn a CelestialCoord into a SkyCoord.

Parameters

celestial
[galsim.CelestialCoord] galsim.CelestialCoord to transform into an astropy.coordinates.SkyCoord

Returns

astropy.coordinates.SkyCoord
SkyCoord corresponding to celestial

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

N

O

P

R

S

T

U

V

W