romanisim

Release 0.0.1.2.dev25+g5eccc1c

romanisim Develeopers

Sep 27, 2022

CONTENTS

A Roman WFI image simulator based on galsim.

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 appropriate PSF for each detector
- realistic world coordinate system
- 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
- reciprocity failure

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" PSF, WCS, and reference files from the Roman CRDS (not yet public!) and webbpsf. 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).

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.

Features not included so far:

- any pedestal/frame 0 features
- non-linear dark features
- realistic treatment of the noise properties in L2 images, using information about how we go from L1 images & resultants to rate images.

1.2 Installation

First, install galsim, paying special attention to the dependence on FFTW. See the documentation here.

Then

    pip install romanisim

and you should be set!

1.3 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 <https://roman-pipeline.readthedocs.io/en/latest/roman/datamodels/metadata.html#metadata>* 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*.

L2 images are currently constructed from count images at present simply by adding the following effects: reciprocity failure, nonlinearity, interpixel capacitance, and read noise. Then the gain is divided out and the image is quantized. The L2 images also do not yet remove the mean dark count rate. Most critically, this process ignores the details of ramp fitting and how that translates into uncertainties on each pixel, but at least the pieces are in place to include that sort of effect. The L2 images should not presently be taken seriously.

1.3.1 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

make_asdf(im[, metadata, filepath])	Wrap a galsim simulated image with ASDF/roman_datamodel metadata.
make_l2(counts, sca[, return_variance, ...])	Simulate an image in a filter given a total count image.
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(sca, targ_pos, date, ...[, ...])	Simulate total counts in a single SCA.

make_asdf

romanisim.image.**make_asdf**(*im*, *metadata=None*, *filepath=None*)

Wrap a galsim simulated image with ASDF/roman_datamodel metadata.

make_l2

romanisim.image.**make_l2**(*counts*, *sca*, *return_variance=False*, *read_noise=None*, *rng=None*, *seed=None*)

Simulate an image in a filter given a total count image.

The idea is that the total count image includes all of the hard work of convolving the scene with the PSF and applying the distortion map, and produces an idealized image of how many photons / electrons land in each pixel. This routine then has the easier job of translating that into an L2 image.

This routine doesn't presently think very hard about what is in an L2 image, and just applies some systematics and noise sources on top. Down the road, we need to think about what the noise really looks like for the slope fits coming out of ramp fitting.

In particular, we just leave the count image the same and add noise; we don't yet make adjustments for the fact that the total exposure time is different from the effective exposure time because at best we only get an exposure time that is the difference of the first and last mean resultant times.

This should get refactored into a separate l2 module.

Parameters

counts
[galsim.Image] total counts in pixel from rate sources in total exposure time

sca
[int] SCA to simulate

return_variance
[bool] If True, also return var_poisson, var_rnoise, var_flat Poisson noise currently does not think about instrumental effects like reciprocity failure, non-linearity, IPC.

read_noise
[ndarray-like[n_pix, n_pix]] read_noise image to use. If None, use galsim.roman.read_noise.

Returns

If return_variance is True, instead a 4-tuple of galsim.Image.
The 4 images represent the observed scene, the Poisson noise, the read noise, and flat field uncertainty-induced noise.

make_test_catalog_and_images

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

This routine kicks the tires on everything in this module.

simulate

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

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

metadata
[dict] metadata structure for Roman asdf file, including information about pointing: metadata['pointing']['ra_v1'], metadata['pointing']['dec_v1'] 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]] 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] 1 or 2, specifying level 1 or level 2 image

rng
[galsim.BaseDeviate] Random number generator to use

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

Returns

asdf structure with simulated image

simulate_counts

romanisim.image.**simulate_counts**(*sca*, *targ_pos*, *date*, *objlist*, *filter_name*, *exptime=None*, *rng=None*, *seed=None*, *ignore_distant_sources=10*, *usecrds=True*, *return_info=False*, *webbpsf=True*, *darkrate=None*)

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

sca
[int] SCA to simulate

targ_pos
[galsim.CelestialCoord or astropy.coordinates.SkyCoord] Location on sky to observe; telescope boresight

date
[astropy.time.Time] Time at which to simulate observation

objlist
[list[CatalogObject]] Objects to simulate

filter_name
[str] Roman filter bandpass to use

exptime
[float] Exposure time to use (if None, default to roman.exptime)

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 of float] dark rate image to use (electrons / s)

Returns

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.

1.4 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.

1.4.1 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
- non-linearity?
- weird non-linear systematics in darks?

Possibly some systematics need to be applied to the individual reads, rather than to the final image. e.g., clearly nonlinearity? 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_read_noise_to_resultants(resultants, tij)	Adds read noise to resultants.
apportion_counts_to_resultants(counts, tij)	Apportion counts to resultants given read times.
ma_table_to_tij(ma_table_number)	Get the times of each read going into resultants for a MA table.
make_asdf(resultants[, filepath, metadata])	Package and optionally write out an L1 frame.
make_l1(counts, ma_table_number[, ...])	Make an L1 image from a counts image.
tij_to_pij(tij)	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_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).

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*)

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. It turns out that Poisson draws are more expensive than binomial draws, too, so it's not clear that that approach offers advantages.

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 resulant.

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

Returns

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

ma_table_to_tij

romanisim.l1.**ma_table_to_tij**(*ma_table_number*)

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

Currently only ma_table_number = 1 is supported, corresponding to a simple fiducial high latitude imaging MA table.

This presently uses a hard-coded, somewhat inflexible MA table description in the parameters file. But that seems like an okay option given that the current 'official' file is slated for redesign when the format is relaxed.

Parameters

ma_table_number
[int] id of multiaccum table to use

Returns

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

make_asdf

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

Returns

roman_datamodels.datamodels.ScienceRawModel for L1 image

make_l1

romanisim.l1.**make_l1**(*counts*, *ma_table_number*, *read_noise=None*, *filepath=None*, *rng=None*, *seed=None*)

Make an L1 image from a counts image.

This apportions the total counts among the different resultants and adds some instrumental effects. The current instrumental effects aren't quite right: nonlinearity and reciprocity failure are applied to the resultants rather than to the reads (which aren't really available to this function).

Parameters

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

ma_table_number
[int] multi accum table number indicating how reads are apportioned among resultants

filepath
[str or None] optional, path to which to write out L1 image

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.

Returns

resultants image array including systematic effects

tij_to_pij

romanisim.l1.**tij_to_pij**(*tij*)

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 conceptually roughly delta_t / sum(delta_t), the fraction of time in each read over the total time, with one important difference. Since we remove the counts that we've already allocated to reads from the number of counts remaining, the probabilities of subsequent reads get scaled up like so that each pij is delta_t / time_remaining.

Parameters

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

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

True if the tij are ascending, otherwise False

1.5 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
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.

1.5.1 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.
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*)

Make a dummy catalog for testing purposes.

Parameters

coord
[galsim.CelestialCoordinate] radius 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*)

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 with keys needed to generate a list of CatalogObject entries for rendering.

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] corresponding to catalog 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] corresponding to catalog

Classes

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

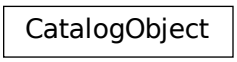
CatalogObject

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.

Class Inheritance Diagram



1.6 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.

1.6.1 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.

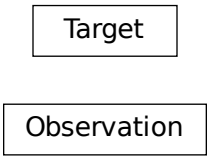
Target

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

Bases: object

A target for observation.

Class Inheritance Diagram



1.7 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 .

1.7.1 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.
get_abflux(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)

get_abflux

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

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 with effective areas for different Roman bandpasses.

1.8 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 spatially constant, achromatic bandpass for each filter when using webbpsf. That is, the PSF is not modeled as varying across each SCA, nor does the PSF 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.

1.8.1 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 PSF variation over the field? Can we be more efficient than making a new InterpolatedImage at each location based on some e.g. quadratic approximation to the PSF's variation with location in an SCA?
- 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_psf(sca, filter_name[, wcs, webbpsf, ...])	Make a PSF profile for Roman.

make_psf

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

Make a PSF profile for Roman.

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

****kw**
[dict] Additional keywords passed to galsim.roman.getPSF

Returns

galsim profile object for convolution with source profiles when rendering scenes.

1.9 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.

1.9.1 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

get_wcs(world_pos[, roll_ref, date, ...])	Get a WCS object for a given sca or set of CRDS parameters.
make_wcs(targ_pos, roll_ref, distortion[, ...])	Create a gWCS from a target position, a roll, and a distortion map.

get_wcs

romanisim.wcs.**get_wcs**(*world_pos*, *roll_ref=0*, *date=None*, *parameters=None*, *sca=None*, *usecrds=True*)

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

Parameters

world_pos
[astropy.coordinates.SkyCoord or galsim.CelestialCoord] boresight of telescope

roll_ref
[float] roll of telescope V3 axis from North to East at boresight

date
[astropy.time.Time] date of observation; used at least is usecrds = None to determine roll_ref.

parameters
[dict] CRDS parameters dictionary specifying appropriate reference distortion map to load.

sca
[int] WFI sensor chip array number

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

Returns

galsim.CelestialWCS for an SCA

make_wcs

romanisim.wcs.**make_wcs**(*targ_pos*, *roll_ref*, *distortion*, *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.

roll_ref
[float] The angle of the V3 axis relative to north, increasing from north to east, at the boresight.

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

Returns

gwcs.wcs object representing WCS for observation

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.

GWCS is initialized via

```
>>> wcs = romanisim.wcs.GWCS(gwcs)
```

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

1.10 Parameters

The parameters module contains a few useful constants describing the Roman telescope. At present these consist only of:

- 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
- the read time.

1.10.1 romanisim.parameters Module

Parameters class storing a few useful constants for Roman simulations.

1.11 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
- flattening & unflattening nested dictionaries.

1.11.1 romanisim.util Module

Miscellaneous utility routines.

Functions

celestialcoord(sky)	Turn a SkyCoord into a CelestialCoord.
flatten_dictionary(d)	Convert a set of nested dictionaries into a flattened dictionary.
random_points_in_cap(coord, radius, nobj[, rng])	Choose locations at random within radius of coord.
scalergb(rgb[, scales, lumrange])	
skycoord(celestial)	Turn a CelestialCoord into a SkyCoord.
unflatten_dictionary(d)	Convert a flattened dictionary into a set of nested dictionaries.

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

flatten_dictionary

romanisim.util.**flatten_dictionary**(*d*)

Convert a set of nested dictionaries into a flattened dictionary.

Some routines want dictionaries of the form dict[key1][key2][key3], while others want dict[key1.key2.key3]. This function converts the former into the latter.

This can do the wrong thing in cases that don't make sense, e.g., if the top level dictionary contains keys including dots that overlap with the names of keys that this function would like to make. e.g.: {'a.b': 1, 'a': {'b': 2}}.

This code is garbage that should be replaced with some better handling of the CRDS <-> ASDF metadata transformations, but I don't fully understand what's happening there and this can stand in as a placeholder.

Parameters

d
[dict] dictionary to flatten

Returns

flattened dictionary, with subdictionaries' keys promoted into the top-level directory with keys adjusted to include dots indicating their former position in the hierarchy.

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

scalergb

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

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

unflatten_dictionary

romanisim.util.**unflatten_dictionary**(*d*)

Convert a flattened dictionary into a set of nested dictionaries.

Some routines want dictionaries of the form dict[key1][key2][key3], while others want dict[key1.key2.key3]. This functions converts the latter into the former.

This code is garbage that should be replaced with some better handling of the CRDS <-> ASDF metadata transformations, but I don't fully understand what's happening there and this can stand in as a placeholder.

Parameters

d
[dict] dictionary to unflatten

Returns

unflattened dictionary, with keys with dots promoted into subdictionaries.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

T

U

V

W