November 10, 2017

GalSim Quick Reference

Contents

1. Overview

The GalSim package provides a number of Python classes and methods for simulating astronomical images. We assume GalSim is installed; see the GalSim Wiki[1] or the file `INSTALL.md` in the base directory for instructions. The package is imported into Python with

```
>>> import galsim
```

and the typical work flow, as demonstrated in the example scripts in the `examples/` directory (all paths given relative to the GalSim base directory from now on), will normally be something like the following:

- Construct a representation of your desired astronomical object as an instance of either the `GSObject` or `ChromaticObject` class, which represent (possibly wavelength-dependent) surface brightness profiles (of galaxies or PSFs). Multiple components can be combined using the special `Add` and `Convolve` functions — see Section 2.
- Apply transformations such as shears, shifts, rotation, or magnification using the methods of the `GSObject` or `ChromaticObject` — see Sections 2.4 & 3.
- Draw the object into a GalSim `Image`, representing a postage stamp image of your astronomical object. This can be done using the `obj.drawImage(...)` method carried by all `GSObjects` and `ChromaticObjects` for rendering images — see Sections 2.4 & 5.
- Add noise to the `Image` using one of the GalSim random deviate classes — see Section 4.
- Add the postage stamp `Image` to a subsection of a larger `Image` instance — see Section 5.2.
- Save the `Image`(s) to file in FITS (Flexible Image Transport System) format — see Sections 5.2 & 6.5.

There are many examples of this workflow in the directory `examples/`, showing most of the GalSim library in action, in the scripts named `demo1.py` – `demo13.py`. An online tutorial guide outlines the capabilities that are demonstrated in the demos. This document provides a brief, reference description of the GalSim classes and methods which can be used in these workflows.

Where possible in the following Sections this document has been hyperlinked to the online GalSim documentation generated by *doxygen*, where a more detailed description can be found. We also suggest accessing the full docstrings for *all* the classes and functions described below in Python itself, e.g. by typing

```
>>> help(galsim.<ObjectName>)
```

within the Python interpreter. If using the *ipython* package, which is recommended, instead simply type

```
In [1]:  galsim.<ObjectName>?
```

and be sure to use the excellent tab-completion feature to explore the many methods and attributes of the GalSim classes.

[1] Note: links to various web pages are colored blue in this document to help you find more information.

2. GSObjects

2.1. GSObject classes and when to use them

GalSim has many classes that represent various types of surface brightness profiles. These are all subclasses of the `GSObject` base class, which encapsulates much of the functionality that is common to all such profiles in GalSim. All but the last two classes listed below are 'simple' `GSObjects` that can be initialized by providing values for their required and optional parameters. The last two are 'compound' classes used to represent combinations of `GSObjects`.

They are summarized in the following hyperlinked list, in which we also give the required parameters for initializing each class in parentheses after the class name. For more information and initialization details for each `GSObject`, the Python docstring for each class is available within the Python interpreter, for example for `Sersic` the documentation would be accessed using

```
>>> help(galsim.Sersic)
```

Alternatively follow the hyperlinks on the class names listed below to view the documentation based on the Python docstrings.

In cases where multiple options for specifying the object *`size`* exist we list these in the object description. We also show some of the optional parameters available for use (e.g. total `flux`) along with default values:

- `galsim.Gaussian(`*`size`*`, flux=1.)`
  A circular 2D Gaussian surface brightness profile. Requires one of the following *`size`* parameters: `sigma`, `fwhm`, or `half_light_radius`.

- `galsim.Moffat(beta, `*`size`*`, flux=1.)`
  A Moffat profile with slope parameter `beta`, used to approximate ground-based telescope PSFs. Requires one of the following *`size`* parameters: `scale_radius`, `fwhm`, or `half_light_radius`. For information about other optional parameters, see the documentation for this object.

- `galsim.Airy(lam_over_diam, obscuration=0., flux=1.)`
  An Airy PSF for ideal diffraction through a circular aperture, parametrized by the wavelength-aperture diameter ratio `lam_over_diam`, with optional `obscuration`.

- `galsim.Kolmogorov(`*`size`*`, flux=1.)`
  A Kolmogorov PSF for long-exposure images through a turbulent atmosphere. Requires one of the following *`size`* parameters: `lam_over_r0`, `fwhm`, or `half_light_radius`.

- `galsim.OpticalPSF(lam_over_diam, flux=1.)`
  A simple model for non-ideal (aberrated) propagation through circular/square apertures, parametrized by the wavelength- aperture dimension ratio `lam_over_diam`, with optional `obscuration`. For information about other optional parameters, see the documentation for this object.

- `galsim.InterpolatedImage(image, ...)`
  A surface brightness profile for which we have an `Image` representation, which is interpolated appropriately to locations other than the pixel centers. For information about other optional parameters, see the documentation for this object.

- `galsim.Pixel(scale, flux=1.)`
  A square tophat profile, used for representing pixels with pixel scale `scale`.

- `galsim.Box(width, height, flux=1.)`
  An arbitrary rectangular box profile.

- `galsim.TopHat(radius, flux=1.)`
  A circular tophat profile.

- `galsim.Sersic(n, half_light_radius, flux=1.)`
  A profile form the Sérsic family of galaxy light profiles, parametrized by an index `n` and `half_light_radius`.

- `galsim.Exponential(`*`size`*`, flux=1.)`
  An Exponential galaxy disc profile, a Sérsic with index `n=1`. Requires one of the following *`size`* parameters: `scale_radius` or `half_light_radius`.

- `galsim.InclinedExponential(inclination, scale_radius, scale_height=0.1*scale_radius, flux=1.)`
  An Inclined Exponential galaxy disc profile. Requires the following parameters: `inclination` and `scale_radius`. `scale_height` can optionally provided; if not, it will default to 0.1*`scale_radius`.

- `galsim.InclinedSersic(n, inclination, scale_radius, scale_height=0.1*scale_radius, trunc=0., flux=1.)`
  An Inclined Sersic galaxy disc profile, and a generalization of the InclinedExponential profile. Requires the following parameters: `n`, `inclination`, and `scale_radius` or `half_light_radius`. `scale_height` can optionally provided; if not, it will default to 0.1*`scale_radius`. Allows a truncation radius `trunc` for the disk.

- `galsim.DeVaucouleurs(half_light_radius, flux=1.)`
  A De Vaucouleurs galaxy bulge profile, a Sérsic with index `n=4` and input `half_light_radius`.

- `galsim.Spergel(nu, `*`size`*`, flux=1.)`
  A class describing a Spergel (2010) profile. Requires one of the following *`size`* parameters: `half_light_radius` or `scale_radius`.

- `galsim.DeltaFunction(flux=1.)`
  A class for describing point sources.

- `galsim.RandomWalk(npoints, half_light_radius, flux=1., ...)`
  A class for generating a set of point sources distributed using a random walk, for representing irregular galaxies or knots of star formation on a disk. The arguments describe how many point sources to distribute and within what spatial extent.

- `galsim.RealGalaxy(real_galaxy_catalog, ...)`
  A galaxy model taken from an image of a real galaxy and includes a correction for the original PSF. Use of this class requires the download of external data. `RealGalaxy` objects can be directly instantiated based on a catalog that is input via the `real_galaxy_catalog` parameter (an instance of the `RealGalaxyCatalog` class), or using the `galsim.COSMOSCatalog` class.

  An example catalog of 100 real galaxies is in the repository itself; a set of real galaxy images, with original PSFs, can be downloaded following instructions on the *RealGalaxy Data Download Page* on the GalSim Wiki, or using an executable `galsim_download_cosmos` that is distributed with GalSim. For information about optional parameters, see the documentation for this object.

- `galsim.ChromaticRealGalaxy(real_galaxy_catalogs, ...)`
  A galaxy model taken from multi-band images of real galaxies, including a correction for the original PSFs. Use of this class requires the download of external data.

- `galsim.Sum( [list of objects] )`
  A compound object representing the sum of multiple `GSObjects`.

- `galsim.Convolution( [list of objects] )`
  A compound object representing the convolution of multiple `GSObjects`.

Note that the last two objects, `Sum` and `Convolution`, are usually created by invoking the `Add` and `Convolve` functions. These functions will automatically create `ChromaticSum` and `ChromaticConvolution` objects instead if any of their arguments are `ChromaticObjects` instead of `GSObects` (see Section 3).

Also note that all of the `GSObjects` except for `RealGalaxy`, `Sum`, and `Convolution` *require* the specification of one size parameter.

2.2. Atmosphere

GalSim can also simulate PSFs generated by turbulent layers in the atmosphere over a wide field of view. The syntax for creating this type of `GSObject` is a little bit different than for those above. First, some number of `AtmosphericScreens` should be assembled into a `PhaseScreenList`:

```
>>> r0_500 = 0.2 # Fried parameter (at 500 nm) in meters
>>> screen_size = 100 # Size of atmospheric screen in meters.
>>> layer0 = galsim.AtmosphericScreen(screen_size=screen_size, r0_500=r0_500)
>>> layer1 = ...
>>> layer2 = ...
>>> psl = galsim.PhaseScreenList([layer0, layer1, layer2])
```

Note that `AtmosphericScreens` have parameters for altitude (which affects how PSFs at different field angles are related) and wind speed (which lets the screens evolve in time over an exposure), in addition to parameters like `r0_500` that control the fixed pattern of turbulence in a screen.

To make a PSF, a telescope aperture is also needed:

```
>>> diam = 4.0 # Meters
>>> obscuration = 0.5 # Linear fractional obscuration
>>> aper = galsim.Aperture(diam=diam, obscuration=obscuration)
```

Then to make PSFs, use the `PhaseScreenList.makePSF` method, optionally specifying a field angle (i.e., where on the focal plane to generate the PSF):

```
>>> exptime = 30.0 # Exposure time in seconds.
>>> lam = 700.0 # Wavelength in nanometers.
>>> psf0 = psl.makePSF(exptime=exptime, lam=lam, aper=aper)
>>> psf1 = psl.makePSF(exptime=exptime, lam=lam, aper=aper,
...                    theta_x=0.1*galsim.arcmin, theta_y=0.1*galsim.arcmin)
```

Note that when working with multiple PSFs coming from the same atmosphere but at different field angles, it can sometimes be faster to first create all of the PSF objects and then draw them, as opposed to interleaving PSF creation and drawing to images.

Also note that for phase screen objects, which include atmospheric PSFs and optical PSFs, drawing images using photon-shooting can be sped up considerably by using a geometric optics approximation, which is enabled for non-optical PSFs. To use a more precise but slower calculation, set the `geometric_shooting` keyword argument to `False` when creating these objects.

2.3. Units

The choice of units for the *size* parameters is up to the user, but it must be kept consistent between all `GSObject`s. These units must also adopted when specifying the `Image` pixel `scale`, whether this is set via the `GSObject` instance method `obj.drawImage(...)` (see Section 2.4), or when constructing the `Image` (see Section 5).

As an example, consider the `lam_over_diam` parameter, which provides an angular scale for the `Airy` via the ratio λ/D for light at wavelength λ passing through a telescope of diameter D. Putting both λ and D in metres and taking the ratio gives `lam_over_diam` in radians, but this is not a commonly used angular scale when describing astronomical objects such as galaxies and stellar PSFs, nor is it often used for image pixel scales. If wishing to use arcsec, which is more common in both cases, the user should multiply the result in radians by the conversion factor $648000/\pi$. In principle, however, any consistent system of units could be used. Users wishing to separately specify the values of λ and D with their natural units of nanometers and meters may do so using `lam` and `diam`; in that case, the keyword `scale_unit` can be used to define the system of units used for the scales in images.

The units for the *flux* of a `GSObject`are nominally photons/s/cm^2, and the units for an image are ADUs (analog-to-digital units). Note, however, that the default exposure time, telescope collecting area, and gain are 1 s, 1 cm^2, and 1 ADU/photon (we combine photon-to-electron and electron-to-ADU efficiencies), respectively, so users who wish to ignore the intricacies of managing exposure times, collecting areas, and gains can simply think of the flux of a `GSObject`in either ADUs or photons.

These details matter more when working with `ChromaticObject`s, where the flux normalization is handled with an `SED` object. In fact, there are two types of `ChromaticObject`s to consider: 1) `ChromaticObject`s representing astronomical objects like stars and galaxies have units (once integrated over angle on the sky) of photons/s/cm^2/nm, 2) `ChromaticObject`s representing wavelength-dependent PSFs are, by contrast, dimensionless. (Technically, this distinction exists for achromatic astronomical objects and PSFs too, but we usually ignore it there.)

2.4. Important GSObject methods

A number of methods are shared by all the `GSObject`s of Section 2. In what follows, we assume that a `GSObject` labelled `obj` has been instantiated using one of the calls described in the documentation linked above. For example,

```
>>> obj = galsim.Sersic(n=3.5, half_light_radius=1.743).
```

Once again, for more information regarding each `galsim.GSObject` method, the Python docstring is available

```
>>> help(obj.<methodName>)
```

within the Python interpreter. Alternatively follow the hyperlinks on the class names above to view the documentation based on the Python docstrings.

Some of the most important and commonly-used methods (shown with parenthesis) or properties (shown without parenthesis) for such an instance are:

- `obj.centroid`
  The (x, y) centroid of the `GSObject` as a `PositionD` (see Section 6.2).
- `obj.flux`
  The flux of the `GSObject`.
- `obj.withScaledFlux(flux_ratio)`
  Return a version of the `GSObject` with the flux scaled by `flux_ratio`.
- `obj.withFlux(flux)`
  Return a version of the `GSObject` with the flux set to `flux`.
- `obj.dilate(scale)`
  Return a version of this `GSObject` with the linear size dilated by a factor `scale`, conserving flux.
- `obj.magnify(mu)`
  Return a version of this `GSObject` with the area magnified by a factor `mu`, conserving surface brightness.
- `obj.shear(...)`
  Return a version of this `GSObject` that has been sheared by some amount. This method can handle a number of different input conventions for the shear (see also `Shear`; Section 6.3). Commonly-used input conventions (supplied as keyword arguments, default values zero):
  - `obj.shear(g1=g1, g2=g2)`
    Apply the first (`g1`) and second (`g2`) component of a shear defined so that $|g| = (a - b)/(a + b)$ where a and b are the semi-major and semi-minor axes of an ellipse.
  - `obj.shear(e1=e1, e2=e2)`
    Apply the first (`e1`) and second (`e2`) component of a shear defined so that $|e| = (a^2 - b^2)/(a^2 + b^2)$ where a and b are the semi-major and semi-minor axes of an ellipse.
  - `obj.shear(g=g, beta=beta)`
    Apply magnitude (`g`) and polar angle (`beta`) of a shear defined using the $|g|$ definition above.
  - `obj.shear(e=e, beta=beta)`
    Apply magnitude (`e`) and polar angle (`beta`) of a shear defined using the $|e|$ definition above.
- `obj.rotate(theta)`
  Return a version of the `GSObject` that has been rotated by an angle `theta` (positive direction anti-clockwise), where `theta` is an `Angle` instance (see Section 6.1).

- `obj.shift(dx, dy)`
  Return a version of the `GSObject` with its centroid position shifted by (dx, dy).

- `image = obj.drawImage(image=None, scale=None, wcs=None, method='auto',`
  `add_to_image=False, ...)`
  Draw and return an `Image` (see Section 5) of the `GSObject`. Some information about important optional parameters (see the linked Python docstrings for more detail), along with default values:

  – `image` (default = `None`)
  If supplied, the drawing will be done into a user-supplied `Image` instance `image`. If not supplied (i.e. `image = None`), an automatically-sized `Image` instance will be returned.

  – `scale` (default = `None`)
  The optional image pixel `scale`, which if provided should use the same units as used for the `GSObject` size parameters.

  – `wcs` (default = `None`)
  The `wcs` may optionally be provided in lieu of a simple pixel scale, in which case this would specify the conversion between image coordinates and world (aka sky) coordinates. The `GSObject` is taken to be defined in world coordinates and this function tells GalSim how to convert to image coordinates when it draws the profile. If neither `scale` nor `wcs` are provided here, then GalSim will use the `wcs` attribute of the `image` if available. Otherwise, it will use the Nyquist scale given the maximum modeled frequency in the `GSObject`.

  – `method` (default = `'auto'`)

    ∗ `'auto'` chooses either `'fft'` or `'real_space'` appropriately based on the kind of profile being drawn.

    ∗ `'fft'` renders the image using a discrete Fourier transform to convolve by the pixel response.

    ∗ `'real_space'` uses direct integration to integrate the flux over the pixels if possible. (It defaults to `'fft'` when such a procedure is impossible or impractical.)

    ∗ `'phot'` renders the image by shooting a finite number of photons. The resulting rendering therefore contains stochastic noise, but uses few approximations. Note however, that you cannot use `'phot'` with a `RealGalaxy` instance. This `method` has a few additional optional parmaeters; see below.

    ∗ `'no_pixel'` does not integrate over the pixel response, but rather samples the profile directly at the pixel centers and multiplies by the pixel area. This is the appropriate choice if your PSF already includes the convolution by the pixel response, e.g. if it comes from an image of a star observed on the same camera.

    ∗ `'sb'` is simlar to `'no_pixel'` except that it does not scale the values by the pixel area. So the drawn values will be direct samples of the surface brightness at each location.

  – `add_to_image` (default = `False`)
  Whether to add flux to a supplied `image` rather than clear out anything in the image before drawing.

  If `method = 'phot'`, there are a number of additional optional parameters. Important examples worthy of mention are:

  – `n_photons` (default = `0`)
  If provided, the number of photons to use. If not provided, use as many photons as necessary to end up with an image with the correct poisson shot noise for the object's `flux`. (Normally, this means use `n_photons=flux`, since `flux` is taken to be in units of photons, but there are exceptions to this.)

- max_extra_noise (default = 0.)
  If provided, the allowed extra noise in each pixel. This is only relevant if n_photons = 0, so the number of photons is being automatically calculated. In that case, if the image noise is dominated by the sky background, you can get away with using fewer shot photons than the full n_photons = flux. Essentially each shot photon can have a flux > 1, which increases the noise in each pixel. The max_extra_noise parameter specifies how much extra noise per pixel is allowed because of this approximation.
- poisson_flux (default = True)
  Whether to allow total object flux scaling to vary according to Poisson statistics for n_photons samples.

The drawImage method has a number of additional optional parameters. Please see the linked Python docstrings for more details.

Finally, you may see by exploring the docstrings that many of the GSObject instances also have their own specialized methods, often for retrieving parameter values. Examples are obj.getSigma() for the Gaussian, or obj.getHalfLightRadius() for many of the GSObjects.

3. Chromaticity

Wavelength-dependent surface brightness profiles are represented as galsim.ChromaticObject instances in GalSim. These objects generally require an galsim.SED to be created, and always require a galsim.Bandpass object in order to draw. Thus we will go over SEDs and Bandpasses first.

3.1. Bandpasses

The galsim.Bandpass class represents a spectral throughput function, which could be an entire imaging system throughput response function (reflection off of mirrors, transmission through filters, lenses and the atmosphere, quantum efficiency of detectors), or individual pieces thereof. Bandpasses, together with spectral energy distributions (SEDs; below) are necessary to compute the relative contribution of each wavelength of a ChromaticObject to a drawn image.

Bandpass instances may be initialized in several ways:

- galsim.Bandpass(filename, wave_type, ...)
  where filename points to a text file with two columns, the first for wavelength and the second for dimensionless throughput. wave_type is used to tell GalSim whether the wavelengths in that file are in units of nanometers or Angstroms.

- galsim.Bandpass(function, wave_type, blue_limit, red_limit, ...)
  where function is a python function that accepts wavelength and returns dimensionless throughput, and wave_type is as above. The keywords red_limit and blue_limit are required in this case to specify the integration limits of the bandpass.

For Bandpass instances initialized from a file, the following two methods can be used to reduce the number of samples used for integrations (and hence reduce the time it takes to draw a ChromaticObject).

- `bandpass.truncate(blue_limit, red_limit, relative_throughput, ...)`
  Return a new bandpass truncated to only include wavelengths between `blue_limit` and `red_limit`, or truncated based on the supplied `relative_throughput` level.

- `bandpass.thin(rel_err,...)`
  Return a thinner version of the bandpass by removing sample values (locations where $F(\lambda)$ is defined) while retaining the accuracy of the integral over the bandpass to the stated relative error `rel_err`. Other aspects of the behavior of this routine are set via additional keyword arguments.

Finally, note that `Bandpass` instances may be multiplied together and are callable, returning dimensionless throughput as a function of wavelength in nanometers.

3.2. SEDs

Spectral energy distributions, respresented by the `SED` class, may be constructed in several ways, similarly to bandpasses:

- `galsim.SED(filename, wave_type, flux_type, ...)`
  where `filename` points to a text file with two columns, the first for wavelength in units given by `wave_type` and the second for flux density in units given by `flux_type`.

- `galsim.SED(function, wave_type, flux_type, ...)`
  where `function` is a python function that accepts wavelength in units given by `wave_type` and returns flux density in units given by `flux_type`.

SEDs can be created using the `astropy.units` module to specify `wave_type` and `flux_type`. For example:

```
>>> from astropy import units as u
>>> sed = galsim.SED(filename, wave_type=u.nm, flux_type=u.erg/(u.s*u.cm**2*u.nm))
```

Important methods for `SED` objects include:

- `sed.withFluxDensity(target_flux_density, wavelength)`
  Return a new `SED` with the same functional shape, but with the flux density (in units of ergs/nm) at the reference wavelength `wavelength` set to `target_flux_density`. Note that `SED` objects are immutable, so the original `SED` is unchanged.

- `sed.calculateFlux(bandpass)`
  Calculate and return the flux transmitted through a `bandpass` in photons.

- `sed.withFlux(target_flux, bandpass)`
  Return a new `SED` for which the flux transmitted through the input `bandpass` is `target_flux`.

- `sed.atRedshift(z)`
  Return a new `SED` instance with wavelength shifted be at redshift z. Note that `SED` instances remember their redshifts (except when created as sums and differences of other SEDs), so applying this method a second twice with the same argument z is equivalent to applying it just once.

GalSim uses SED instances both to represent spectral density and dimensionless wavelength-dependent normalization. (The former for stars and galaxies, the latter for chromatic PSFs.) SED instances can be added together, multiplied by scalars or functions (of wavelength in nanometers), and are callable, returning either flux density in photons/s/cm^2/nm or dimensionless normalization as appropriate. Some operations are restricted depending on SED.flux_type. For instance, the product of two SEDs with units photons/s/cm^2/nm is prohibited since the resulting object would not have the dimensions of an SED.

3.3. ChromaticObjects

Chromatic surface brightness profiles are generally constructed by modifying an existing GSObject. The simplest possible ChromaticObject can be formed by passing a GSObject to the ChromaticObject constructor:

```
>>> obj = galsim.Gaussian(fwhm=1.0)
>>> chromatic_obj = galsim.ChromaticObject(obj)
```

At this stage, chromatic_obj represents the same profile as obj, but now has access to ChromaticObject methods.

The simplest way to construct a non-trivial chromatic object is to multiply a GSObject by an SED. This creates a separable wavelength-dependent surface brightness profile $\mathrm{I}(x, y, \lambda) = \mathrm{I}_0(x, y)\mathrm{f}(\lambda)$:

```
>>> chromatic_obj = obj * sed
```

ChromaticObjects may be combined and transformed similarly to GSObjects, using the functions and methods Add, Convolve, withScaledFlux, dilate, magnify, shear, rotate, and shift.

The transformation methods of ChromaticObjects, like dilate and shift, can also accept as an argument a function of wavelength (in nanometers) that returns a wavelength-dependent dilation, shift, etc. These can be used to implement chromatic PSFs. For example, a diffraction limited PSF might look like:

```
>>> psf500 = galsim.Airy(lam_over_diam=2.0)
>>> chromatic_psf = ChromaticObject(psf500).dilate(lambda w:(w/500)**(1.0))
```

The drawImage method of a ChromaticObject is similar to the drawImage method of a GSObject, except that it requires a Bandpass object as its first argument. Note that only ChromaticObjects with SEDs having units of photons/s/cm^2/nm can be drawn.

```
>>> gband = galsim.Bandpass(lambda w:1.0, wave_type='nm', blue_limit=410, red_limit=550)
>>> image = chromatic_obj.drawImage(gband)
```

GalSim comes with built-in support for ground-based PSFs affected by differential chromatic refraction and Kolmogorov chromatic seeing (FWHM $\propto \lambda^{-0.2}$) through the following class:

- ChromaticAtmosphere(base_obj, base_wavelength, ...)
  where base_obj is the fiducial PSF at wavelength base_wavelength. There are a number of options for specifying details like the position of the object with respect to zenith and the physical properties of the atmosphere.

There are also classes for representing chromatic optical PSFs:

- `ChromaticAiry(...)`
  for representing a chromatic Airy profile, where the user must specify either `lam` (the telescope diameter in meters) or `lam_over_diam` (the ratio of some fiducial wavelength to the diameter).

- `ChromaticOpticalPSF(...)`
  for representing a fully chromatic optical PSF, including chromatic aberrations.

For sufficiently complex chromatic objects, an optional optimization can be invoked to speed up the image rendering process; for more information, view the docstring for the `setupInterpolation` method associated with `ChromaticObjects`.

4. Random deviates

4.1. Random deviate classes and when to use them

Random deviates can be used to add a stochastic component to the modeling of astronomical images, such as drawing object parameters according to a given distribution or generating random numbers to be added to image pixel values to model noise.

We now give a short summary of the 8 random deviates currently implemented in GalSim. The optional parameter *`seed`* listed below is used to seed the pseudo-random number generator: it can either be omitted (the random deviate seed will be set using the current time), set to an integer seed, or used to pass another random deviate (the new instance will then use and update the same underlying generator as the input deviate).

The deviates, with a description of their distributions, parametrization and default parameter values, are as follows:[2]

- `galsim.UniformDeviate(`*`seed`*`)`
  Uniform distribution in the interval $[0, 1)$.

- `galsim.GaussianDeviate(`*`seed`*`, mean=0., sigma=1.)`
  Gaussian distribution with `mean` and standard deviation `sigma`.

- `galsim.BinomialDeviate(`*`seed`*`, N=1, p=0.5)`
  Binomial distribution for `N` trials each of probability `p`.

- `galsim.PoissonDeviate(`*`seed`*`, mean=1.)`
  Poisson distribution with a single `mean` rate.

- `galsim.WeibullDeviate(`*`seed`*`, a=1., b=1.)`
  Weibull distribution family (includes Rayleigh and Exponential) with shape parameters `a` and `b`.

[2] Note that, with the exception of `DistDeviate`, all the deviate classes are defined in the C++ layer, so the hyper-links below link to that code, rather than the Python documentation. The syntax is not too different in Python, but unfortunately, the docstrings are not currently linkable directly. For more information, please refer to the full docstrings, accessible via commands like `help(galsim.UniformDeviate)` in the Python interpreter.

- `galsim.GammaDeviate(`*`seed`*`, alpha=1., beta=1.)`
  Gamma distribution with parameters `alpha` and `beta`.

- `galsim.Chi2Deviate(`*`seed`*`, n=1.)`
  χ^2 distribution with degrees-of-freedom parameter `n`.

- `galsim.DistDeviate(`*`seed`*`, function, x_min, x_max)`
  Use an arbitrary function for $P(x)$ from `x_min` .. `x_max`.

It is possible to specify the random seed so as to get fully deterministic behavior of the noise when running a particular script.

4.2. Important random deviate methods

We now illustrate the most commonly-used methods of the random deviates, assuming that some random deviate instance `dev` has been instantiated, for example by

```
>>> dev = galsim.GaussianDeviate(sigma=3.9, mean=50.).
```

The most important and commonly-used method for such instances is:

- `dev()`
  Calling the deviate directly simply returns a single new random number drawn from the distribution represented by `dev`. As an example:

  ```
  >>> dev = galsim.UniformDeviate(12345)
  >>> dev()
  0.9296160866506398
  >>> dev()
  0.8901547130662948
  ```

4.3. Noise models

One common way to use the random deviates is as part of a noise model for adding noise to an image. These have their own separate hierarchy of classes

- `galsim.GaussianNoise(rng, sigma=1.)`
  Every pixel gets Gaussian noise with rms `sigma`, using the random number generator given as `rng`, which may be any deviate instance.

- `galsim.PoissonNoise(rng, sky_level=0.)`
  Every pixel gets Poisson noise according to the flux in the image plus an option sky level, `sky_level`, using the random number generator given as `rng`, which may be any deviate instance.

- `galsim.DeviateNoise(dev)`
  The noise value for every pixel is drawn from the given Deviate instance `dev`.

- `galsim.CCDNoise(rng, sky_level=0., gain=1., read_noise=0.)`
  A combination of Poisson noise (with a gain value in electrons/ADU) and Gaussian read noise, using the random number generator given as `rng`, which may be any deviate instance.

- `galsim.VariableGaussianNoise(rng, var_image)`
  Every pixel gets Gaussian noise with variance given by the `var_image` parameter (an `Image` instance), using the random number generator given as `rng`, which may be any deviate instance. The supplied `var_image` must be the same size and shape as the `Image` onto which the noise is eventually added.

- `galsim.CorrelatedNoise(image, rng, ...)`
  The provided image should be a pure noise field (i.e. a blank area of sky with no detectable objects). The correlated noise of this image will be computed, which can then be applied to any other image, laying down noise with the same correlations. The provided random number generator, `rng`, which may be any deviate instance will be used as the basis of the required random numbers..

To apply noise to an `Image` using these noise models, the command is simply:

- `image.addNoise(noise)`
  This adds stochastic noise, according to the noise model `noise`, to each element of the data array in the `Image` instance `image`.

In addition, GalSim can remove some or all of the noise correlations in an image that has a known correlated noise:

- `image.whitenNoise(noise)`
  This adds extra correlated noise to the image such that if the image initially has correlated noise given by `noise` (which should be a `CorrelatedNoise` object), then the image will end up with white noise. It tries to add the minimum amount of extra noise possible to achieve this.

- `image.symmetrizeNoise(noise, order=4)`
  This is similar to `whitenNoise`, except that rather than making the resulting noise profile completely white, it just makes the final noise in the image have N-fold symmetry where N is given by `order`. This typically requires much less noise to be added than `whitenNoise`, and for many purposes this is sufficient to remove the adverse effects of correlated noise.

5. Images

5.1. Image classes and when to use them

The GalSim `Image` class stores array data, along with the bounds of the array, and a function that converts between image coordinates and world coordinates (also known as sky coordinates). The most common World Coordinate System (WCS) function that you will encounter is a simple scaling of the units from pixels to arcsec. This WCS can be specified simply by `im.scale`, as we have seen already. More complicated WCS functions would need to be referenced via `im.wcs`. See the docstring for `BaseWCS` for more details.

`Image` instances can be operated upon to add stochastic noise simulating real astronomical images (see Section 4), and have methods for writing to FITS format output.

The most common way to initialize an image is with two integer parameters `nx` and `ny`, giving the image extent in the x and y dimensions, respectively. Example initialization is therefore:

- `galsim.Image(nx, ny)`

This would create an image with single precision (32 bit) floats for the data elements, which is usually the most appropriate type for astronomical images. However, you can specify other types for the data using a suffix letter after `Image`:

- `galsim.ImageS(nx, ny)` for 16 bit integers.
- `galsim.ImageI(nx, ny)` for 32 bit integers.
- `galsim.ImageF(nx, ny)` for single precision (32 bit) floats.
- `galsim.ImageD(nx, ny)` for double precision (64 bit) floats.

Other ways to construct an `Image` can be found in the docstrings, including via an input NumPy array.

To access the data as a NumPy array, simply use the `image.array` attribute, where `image` is an instance of one of these `Image` classes. However, note that the individual elements in the array attribute are accessed as `image.array[y,x]`, matching the standard NumPy convention, while the `Image` class's own accessors are all (x, y) in ordering.

5.2. Important Image methods and operations

We now illustrate the most commonly-used methods of `Image` class instances. We will assume that some `Image` instance `image` has been instantiated, for example by

```
>>> image = galsim.Image(100, 100).
```

This `Image` instance is then ready to pass to a `GSObject` for drawing. The most important and commonly-used methods for such an instance are:

- `image.bounds`
  Get the bounding box of the data.
- `image.wcs`
  Get or set the WCS function to convert between image coordinates and world coordinates.

- `image.scale`
  Get or set the pixel scale `scale` for this image. The getter only works if the WCS is really just a pixel scale. The setter will make it a pixel scale.

- `image.addNoise(noise)`
  This adds stochastic noise according to the given noise model, `noise` to each element of the data array in `image`. This is the method previously referenced in Section 4.

- `image.write(file_name, ...)`
  Write the `image` to a FITS file, given by `file_name`. There are other options for how to write this (such as writing to an HDUList rather than a file), so see the docstring for more details, including details about the optional parameters. In Section 6.5 we discuss how to write to multi-extension FITS files.

`Image` instances are also returned when accessing a sub-section of an existing `Image`. For example

```
>>> imsub = image.subImage(bounds)
```

where `bounds` is a `BoundsI` instance (see Section 6.2) assigns `imsub` as an view into the sub-region of `image` lying in the area represented by `bounds`. Equivalent syntax is also

```
>>> imsub = image[bounds]
```

It is also possible to change the values of a sub-region of an image this way, for example

```
>>> image[imsub.bounds] += imsub
```

if wishing to add the contents of `imsub` to the area lying within its bounds in `image`. Note that here we have made use of the `image.bounds` attribute carried by all of the `Image` classes.

6. Miscellaneous classes and functions

A summary of miscellaneous GalSim library objects, subcategorized into broad themes. As ever, docstrings for *all* the classes and functions below can be accessed via `help(galsim.<Name>)` within the Python interpreter.

6.1. Angles

- `galsim.Angle(value, angle_unit)`
  Class to represent angles (with multiple unit types), which can be initialized by multiplying a numerical value and an `AngleUnit` instance `angle_unit` (see below).

- `galsim.AngleUnit`
  There are five built-in `AngleUnits` which are always available for use:

  - `galsim.radians`
  - `galsim.degrees`
  - `galsim.hours`

   - `galsim.arcmin`
   - `galsim.arcsec`

  Please see the Python docstrings for information about defining your own `AngleUnits`.

6.2. Bounds and Positions

- `galsim.BoundsI(...)`
  `galsim.BoundsD(...)`
  Classes to represent image boundaries as the vertices of a rectangle.

- `galsim.PositionI(x, y)`
  `galsim.PositionD(x, y)`
  Classes to represent 2D positions on the `x`-`y` plane, e.g., for describing object centroid positions.

  For both bounds and positions, the `I` and `D` refer to integer and double-precision floating point representations.

6.3. Shears

- `galsim.Shear(...)`
  Class to represent shears in a variety of ways. This class can be initialized using a variety of different parameter conventions (see the doc string for the complete list). Commonly-used examples (supplied as keyword arguments, default values zero):

   - `galsim.Shear(g1=g1, g2=g2)`
     set via the first (`g1`) and second (`g2`) component of a shear defined so that $|g| = (a - b)/(a + b)$ where a and b are the semi-major and semi-minor axes of an ellipse.
   - `galsim.Shear(e1=e1, e2=e2)`
     set via the first (`e1`) and second (`e2`) component of a shear defined so that $|e| = (a^2 - b^2)/(a^2 + b^2)$ where a and b are the semi-major and semi-minor axes of an ellipse.
   - `galsim.Shear(g=g, beta=beta)`
     set via magnitude (`g`) and polar angle (`beta`) of a shear defined according to the $|g|$ definition above.
   - `galsim.Shear(e=e, beta=beta)`
     set via magnitude (`e`) and polar angle (`beta`) of a shear defined according to the $|e|$ definition above.

6.4. Lensing shear fields

GalSim has functionality to simulate scientifically-motivated lensing shear fields. The two relevant classes for users are:

- `galsim.PowerSpectrum(...)`
  Represents a flat-sky shear power spectrum $P(k)$, where the E and B-mode power spectra can be separately

specified as `E_power_function` and `B_power_function`. The `getShear(...)` method is used to generate a random realization of a shear field from a given `PowerSpectrum` object, and there are methods to get convergence or magnification as well.

- `galsim.NFWHalo(...)`
  Represents a matter density profile corresponding to a projected, circularly-symmetric NFW profile such as might be used to simulate lensing by a galaxy cluster. This class has two methods of interest for users, `getShear()` and `getConvergence()`, which can be used to get the shears and convergences at *any* (non-gridded) image-plane position.

These classes have additional requirements on the units used to specify positions; see the documentation for these classes for more details.

The GalSim repository also contains a module with a `PowerSpectrumEstimator` class that can be used to estimate shear power spectra from gridded shear values. See the linked documentation for more information.

6.5. Additional FITS input/output tools

For all of these functions, there are also options for reading/writing directly from/to an HDUList rather than a file, so see the linked doc strings for more details. Optional parameters control how to handle compression, although the default in all cases is to try to use the correct compression scheme based on the file's extension.

- `image = galsim.fits.read(file_name, ...)`
  Returns an `Image` instance `image` from a FITS file, `file_name`. If the FITS file has a WCS defined in the header, then GalSim will attempt to read that WCS and store it as `image.wcs`.

- `image_list = galsim.fits.readMulti(file_name, ...)`
  Returns a Python `list` of `Image` instances (`image_list`) from a Multi-Extension FITS file, `file_name`.

- `galsim.fits.writeMulti(image_list, file_name, ...)`
  Write multiple `Image` instances stored in a Python `list` (`image_list`) to a Multi-Extension FITS file, `file_name`.

- `galsim.fits.writeCube(image_list, file_name, ...)`
  Write multiple `Image` instances stored in a Python `list` (`image_list`) to a three-dimensional FITS datacube, `file_name`.