Draft March 4, 2014

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* or the file `INSTALL.md` in the base directory `/your/path/to/GalSim/` 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 `/your/path/to/GalSim/` 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.
- Chromatic objects will generally also require the construction of spectral energy distribution `SED` class instances and `Bandpass` class instances.
- Apply transformations such as shears, shifts or magnification using the methods of the `GSObject` or `ChromaticObject` — see Section 2.3.
- Draw the object into a GalSim `Image`, representing a postage stamp image of your astronomical object. This can be done using the `obj.draw(...)` or `obj.drawShoot(...)` methods carried by all `GSObject`s or the `obj.draw` method carried by `ChromaticObject`s for rendering images — see Sections 2.3 & 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 — or to a Python `list` containing multiple `Image` instances.
- 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` – `demo12.py`. 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.

2. GSObjects

2.1. GSObject classes and when to use them

There are currently 13 types of `GSObjects` that represent various types of surface brightness profiles. The first 11 listed 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.

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

- `galsim.Gaussian(`*`size`*`, flux=1.)`
  a 2D Gaussian light profile. Requires one of the following *`size`* parameters to be set as a keyword argument: `sigma`; `fwhm`; `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 to be set as a keyword argument: `scale_radius`; `fwhm`; `half_light_radius`. For information about other optional parameters, see the documentation for this object.

- `galsim.AtmosphericPSF(`*`size`*`, flux=1.)`
  currently simply an image-based implementation of a Kolmogorov PSF (see below), and therefore deprecated, but expected to evolve to store a stochastically modeled atmospheric PSF in the near

future. Requires one of the following *size* parameters to be set as a keyword argument: `fwhm`; `lam_over_r0`. 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.)`
  the Kolmogorov PSF for long-exposure images through a turbulent atmosphere. Requires one of the following *size* parameters to be set as a keyword argument: `lam_over_r0`; `fwhm`; `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 class representing in principle arbitrary surface brightness profiles for which we have an Image representation. For information about other optional parameters, see the documentation for this object.

- `galsim.Pixel(scale, flux=1.)`
  used for integrating light onto square pixels.

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

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

- `galsim.Exponential(`*size*`, flux=1.)`
  the Exponential galaxy disc profile, a Sérsic with index `n=1`. Requires one of the following *size* parameters to be set as a keyword argument: `scale_radius`; `half_light_radius`.

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

- `galsim.RealGalaxy(real_galaxy_catalog, ...)`
  models galaxies using real data, including a correction for the original PSF. Requires the download of external data, stored and input as the `real_galaxy_catalog` parameter (an instance of the `RealGalaxyCatalog` class), for full functionality.

  An example catalog of 100 real galaxies is in the repository itself; a set of ~26 000 real galaxy images, with original PSFs, can be downloaded from the *RealGalaxy Data Download Page* on the

GalSim Wiki:
https://github.com/GalSim-developers/GalSim/wiki/RealGalaxy%20Data%20Download%20Page.
For information about other optional parameters, see the documentation for this object.

- `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 `galsim.Add` and `galsim.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`, `Add`, and `Convolve` *require* the specification of one radius size parameter.

2.2. Units

The choice of units for the size parameters is up to the user, but it must be kept consistent between all `GSObjects`. These units must also adopted when specifying the `Image` pixel `scale`, whether this is set via the `GSObject` instance methods `obj.draw(...)` and `obj.drawShoot(...)` (see Section 2.3), or when setting the scale of an `Image` using `image.scale = scale` (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.

2.3. Important GSObject methods

A number of methods are shared by all the `GSObjects` of Section 2, and are also to be found in `galsim/base.py` within the definition of the `GSObject` base class. 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).
```

One important fact about `GSObjects` is that all of the methods which change the properties of the astronomical object represented by the instance (e.g., `setFlux()`, `applyShear()` etc.) also make fundamental changes to the instance itself. In most cases this will mean that special methods available to individual classes described in Section 2.1, such as `getFWHM()` for the `Moffat`, will be unavailable.

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 for such an instance are:

- `obj.copy()`
  return a copy of the `GSObject`.
- `obj.centroid()`
  return the (x, y) centroid of the `GSObject` as a `PositionD` (see Section 6.2).
- `obj.getFlux()`
  get the flux of the `GSObject`.
- `obj.scaleFlux(flux_ratio)`
  multiply the flux of the `GSObject` by `flux_ratio`.
- `obj.setFlux(flux)`
  set the flux of the `GSObject` to `flux`.
- `obj.applyTransformation(ellipse)`
  apply an `Ellipse` transformation represented by `ellipse` to the `GSObject` (see `Ellipse`; Section 6.3).
- `obj.applyDilation(scale)`
  change of the linear size of the `GSObject` by a factor `scale`, conserving flux.
- `obj.applyMagnification(scale)`
  dilate linear size by `scale` and multiply total flux by `scale`^{2}, conserving surface brightness.
- `obj.applyShear(...)`
  apply a shear to the `GSObject`, handling a number of different input conventions (see also `Shear`; Section 6.3). Commonly-used input conventions (supplied as keyword arguments, default values zero):

- `obj.applyShear(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.applyShear(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.applyShear(g=g, beta=beta)`
  apply magnitude (`g`) and polar angle (`beta`) of a shear defined using the $|g|$ definition above.
- `obj.applyShear(e=e, beta=beta)`
  apply magnitude (`e`) and polar angle (`beta`) of a shear defined using the $|e|$ definition above.

◦ `obj.applyRotation(theta)`
apply a rotation of `theta` (positive direction anti-clockwise) to the `GSObject`, where `theta` is an `Angle` instance (see Section 6.1).

◦ `obj.applyShift(dx, dy)`
apply a (dx, dy) position shift to the `GSObject` centroid.

◦
```
image = obj.draw(image=None, scale=None, wcs=None,
                 add_to_image=False, ...)
```
draw and return an `Image` (see Section 5) of the `GSObject` using Discrete Fourier Transforms and interpolation to perform the image rendering. Note that if a profile is not convolved with a pixel response before drawing, then `draw` samples the surface brightness distribution without integrating within pixels, so the sum of pixel values might not equal the GSObject flux. 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`.

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

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

- image = obj.drawShoot(image=None, scale=None, wcs=None add_to_image=False, ...)
  draw and return an Image (see Section 5) of the GSObject by shooting a finite number of photons. The resulting rendering therefore contains stochastic noise, but uses few approximations. Note however, that you cannot drawShoot with a RealGalaxy instance. drawShoot shares all the parameters listed for draw, above, but the drawShoot method also has 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.
  - 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.

  As before, you are strongly encouraged to 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 in GalSim as galsim.ChromaticObjects. 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. `Bandpass`es, 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`es may be constructed in several ways:

- `galsim.Bandpass(filename)`
  where `filename` points to a text file with two columns, the first for wavelength and the second for dimensionless throughput.

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

- `galsim.Bandpass(expression, red_limit=red_limit, blue_limit=blue_limit)`
  where `expression` is a string that can be evaluated into a python function via
  `eval('lambda wave : '+expression)`,
  e.g. `expression = '0.8 + 0.2 * (wave-800)'`. In this case, `red_limit` and `blue_limit` are required to specify the integration limits of the bandpass.

By default, the units for wavelength in the above functions/file are assumed to be nanometers. If the keyword argument `wave_type = 'Ang'` is supplied, then the wavelengths will instead be interpretted as Angstroms.

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=blue_limit, red_limit=red_limit, relative_throughput=relative_throughput)` Clip the wavelength range to be between `blue_limit` and `red_limit`. Additionally clip any leading or trailing wavelengths for which the throughput is less than the fraction `relative_throughput` of the peak throughput.

- `bandpass.thin(rel_err)` Remove samples defining the bandpass while retaining the accuracy of the integral over the bandpass to the stated relative error `rel_err`.

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

3.2. SEDs

Spectral energy distributions may be constructed in several ways, similarly to bandpasses:

- `galsim.SED(filename)`
  where `filename` points to a text file with two columns, the first for wavelength in nanometers and the second for flux density.
- `galsim.SED(function)`
  where `function` is a python function that accepts wavelength in nanometers and returns flux density.
- `galsim.SED(expression)`
  where `expression` is a string that can be evaluated into a python function via
  `eval('lambda wave : '+expression)`,
  e.g. `expression = '0.8 + 0.2 * (wave-800)'`.

The units for wavelength in the above constructions can be set to Angstroms by supplying the keyword argument `wave_type = 'Ang'`. The units for flux density in the above are assumed to be proportional to ergs/nm, but can be overridden to be proportional to ergs/Hz by setting `flux_type = 'fnu'`, or overridden to be proportional to photons/nm by setting `flux_type = 'fphotons'`.

Important methods for `SED` objects include:

- `SED.withFluxDensity(target_flux_density, base_wavelength)`
  Return a new `SED` with flux density (in units proportional to ergs/nm) at wavelength `base_wavelength` set to `target_flux_density`. Note that `SED` objects are immutable, so the original `SED` is unchanged.
- `calculateFlux(bandpass)` Calculate and return the flux transmitted through a `Bandpass` in photons.
- `withFlux(target_flux, bandpass)` Return a new `SED` with transmitted flux through `Bandpass` equal to `flux_norm`.
- `atRedshift(z)` Return a new `SED` with wavelength shifted be at redshift z. Note that `SED`s 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.

Finally, note that `SED`s can be added together, multiplied by scalars or functions (of wavelength in nanometers), and are callable, returning flux density in photons/nm.

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` essentially 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:

$$\text{ChromaticObject}(x, y, \lambda) = \text{GSObject}(x, y)\text{SED}(\lambda) \tag{1}$$

```
>>> gal = galsim.Sersic(n=2.5, half_light_radius=1.1)
>>> SED = galsim.SED('wave**1.1') # Power-law spectrum.
>>> chromatic_gal = gal*SED
```

`ChromaticObjects` may be combined and transformed similarly to `GSObjects`, using the functions and methods `Add`, `Convolve`, `scaleFlux`, `applyExpansion`, `applyDilation`, `applyMagnification`, `applyShear`, `applyLensing`, `applyRotation`, `applyShift`, and all the `create*` methods as well.

The `applyDilation`, `applyExpansion`, and `applyShift` methods of `ChromaticObjects` can also accept as an argument a function of wavelength (in nanometers) that returns a wavelength-dependent dilation, expansion, or shift. 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)
>>> chromatic_psf.applyDilation(lambda w:(w/500.0)**(1.0))
```

The draw method of a `ChromaticObject` is similar to the draw method of a `GSObject`, except that it requires a `Bandpass` object as its first argument.

```
>>> gband = galsim.Bandpass(lambda w:1.0, blue_limit=410, red_limit=550)
>>> pix = galsim.Pixel(0.2)
>>> final = galsim.Convolve(chromatic_gal, chromatic_psf, pix)
>>> image = final.draw(gband)
```

GalSim also 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 function:

- `ChromaticAtmosphere(base_obj, base_wavelength, zenith_angle, position_angle=position_angle)`:
  Here `base_obj` is the fiducial PSF at wavelength `base_wavelength`. Differential chromatic refraction is calculated for a telescope pointed at `zenith_angle`, where the zenith lies in the direction `position_angle` measured from "up" through "right".

For example:

```
>>> psf500 = galsim.Kolmogorov(fwhm=0.67)
>>> psf = galsim.ChromaticAtmosphere(psf500, 500, zenith_angle=30*galsim.degrees)
```

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 9 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:

- `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`.

- `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. Unfortunately the random deviate classes are not yet fully integrated within the documentation, due to their being C++ with compiled Python wrappers. This means that the class names above and methods below are not yet hyperlinked. For more information, please refer to the full docstrings in `galsim/random.py`, or type

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

within the Python interpreter.

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(dev, sigma=1.)`
  Every pixel gets Gaussian noise with rms `sigma`, using the same random number generator as the supplied Deviate instance `dev`.
- `galsim.PoissonNoise(dev, sky_level=0.)`
  Every pixel gets Poisson noise according to the flux in the image plus an option sky level, `sky_level`, using the same random number generator as the supplied Deviate instance `dev`.
- `galsim.DeviateNoise(dev)`
  The noise value for every pixel is drawn from the given Deviate instance `dev`.
- `galsim.CCDNoise(dev, 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 same random number generator as the supplied Deviate instance `dev`.

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`.

5. Images

5.1. Image classes and when to use them

The GalSim `Image` classes store 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 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.

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.

Unfortunately the `Image` classes are not yet fully integrated within the online documentation, due to their being in C++ with compiled Python wrappers. This means that the class names above and methods below are not hyperlinked. However, the full docstrings are available in `galsim/image.py`, so please refer there for more information, or type

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

within the Python interpreter.

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.ImageD(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 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(dev)`
  this adds stochastic noise, distributed as represented by the random deviate instance `dev`, to each element of the data array in `image`. This is the method previously referenced in Section 4.

- `image.write(fits, ...)`
  write the `imageView` to a FITS file or object as determined by the `fits` input parameter (see `galsim/fits.py`). 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, and `galsim/angle.py`).

- `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 (see `galsim/bounds.py`).

- `galsim.PositionI(x, y)`
  `galsim.PositionD(x, y)`
  classes to represent 2D positions on the `x`-`y` plane (see `galsim/position.py`), 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. Shear and Ellipse transformations

- `galsim.Shear(...)`
  class to represent shears in a variety of ways. This class can be initialized using a variety of different parameter conventions (see `galsim/shear.py`). 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.

- galsim.Ellipse(...)
  class to represent ellipses and thus ellipse-type transformations, specifically shears, shifts, and dilations. The class can be initialized using a variety of different parameter conventions (see, e.g., galsim/ellipse.py), including being initialized with a Shear instance.

6.4. Lensing shear fields

GalSim has functionality to simulate scientifically-motivated lensing shear fields. The code and documentation for the "lensing engine" is in galsim/lensing.py. 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 even if GalSim is not installed: galsim/pse.py (see documentation in that file for more information).

6.5. Additional FITS input/output tools

- image = galsim.fits.read(fits)
  returns an Image instance image from a FITS representation fits. If fits is a string it is interpreted as a filename, otherwise it is interpreted as a PyFITS representation of HDU data (see galsim/fits.py). 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(fits)`
  returns a Python `list` of `Image` instances (`image_list`) from a Multi-Extension FITS file or PyFITS HDU object, specified by the `fits` input parameter (see `galsim/fits.py`).

- `galsim.fits.writeMulti(image_list, fits, ...)`
  write multiple `Image` instances stored in a Python `list` (`image_list`) to a Multi-Extension FITS file or PyFITS HDU object, specified by the `fits` input parameter (see `galsim/fits.py`).

- `galsim.fits.writeCube(image_list, fits, ...)`
  write multiple `Image` instances stored in a Python `list` (`image_list`) to a three-dimensional FITS datacube or PyFITS HDU object, specified by the `fits` input parameter (see `galsim/fits.py`).

The routines for reading and writing FITS images are able to handle compressed inputs / outputs via keywords.