Draft September 18, 2012

GalSim Quick Reference

Contents

1. Overview

The GalSim package provides a number of Python classes and methods for simulating astronomical images. We assuming 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 the `GSObject` class, which represent surface brightness profiles (of galaxies or PSFs). Multiple components can be combined using the special `Add` and `Convolve` classes — see Section 2.
- Apply transformations such as shears, shifts or magnification using the methods of the `GSObject` — 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 for rendering images — see Sections 2.3 & 4.
- Add noise to the `Image` using one of the GalSim random deviate classes — see Section 3.
- Add the postage stamp `Image` to a subsection of a larger `Image` instance — see Section 4.2 — or to a Python `list` contaning multiple `Image` instances.
- Save the `Image`(s) to file in FITS (Flexible Image Transport System) format — see Sections 4.2 & 5.4.

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` – `demo8.py`.

We now provide a brief, reference description of the GalSim classes and methods which can be used in this workflow. 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 the *all* the classes and functions descrbed below in Python itself, e.g. by typing

```
>>> print galsim.<ObjectName>.__doc__
```

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

```
>>> print galsim.Sersic.__doc__
```

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

In the order in which the classes appear in `galsim/base.py`:

- `galsim.Gaussian(...)`
  a 2D Gaussian light profile.

- `galsim.Moffat(beta, ...)`
  a Moffat profile with slope parameter `beta`*, used to approximate ground-based telescope PSFs.*

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

- `galsim.Airy(lam_over_diam, ...)`
  an Airy PSF for ideal diffraction through a circular aperture, parameterized by the wavelength-aperture diameter ratio `lam_over_diam`*, with optional obscuration.*

- `galsim.Kolmogorov(...)`
  the Kolmogorov PSF for long-exposure images through a turbulent atmosphere.

- `galsim.OpticalPSF(lam_over_diam, ...)`
  a simple model for non-ideal (aberrated) propagation through circular or square apertures, parameterized by the wavelength-aperture dimension ratio `lam_over_diam`*, with optional obscuration.*

- `galsim.Pixel(xw, ...)`
  used for integrating light onto square or rectangular pixels, requires at least one side dimension `xw`*.*

- `galsim.Sersic(n, ...)`
  the Sérsic family of galaxy light profiles, parameterized by an index `n`*.*
- `galsim.Exponential(...)`
  the Exponential galaxy disc profile, a Sérsic with index `n=1`*.*
- `galsim.DeVaucouleurs(...)`
  the De Vaucouleurs galaxy bulge profile, a Sérsic with index `n=4`*.*
- `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 in `real_galaxy_catalog` *(an instance of the* `RealGalaxyCatalog` *class), for full functionality.*
- `galsim.Add(...)`
  a compound *object representing the sum of multiple* `GSObjects`*.*
- `galsim.Convolve(...)`
  a compound *object representing the convolution of multiple* `GSObjects`*.*

Note that all of the `GSObjects` except for `RealGalaxy`, `Add`, and `Convolve` *require* the specification of some radius parameter, where the choice of possible radii to specify (e.g., half-light radius, FWHM, etc.) is given in the documentation for the class.

2.2. Units

The choice of units for these size specifications is up to the user, but it must be kept consistent between all `GSObjects`. These units must also adopted when specifying the `Image` sample rate `dx`, 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` with a given `dx` using the `image.setScale(dx)` method (see Section 4).

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.[1]

---

[1]Unfortunately, as it happens, there is currently an issue in the use of `OpticalPSF` with very small numerical values of `lam_over_diam`, such as those which would be needed if using radians as the system of units. This will hopefully be fixed soon.

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 representend 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.

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 5.2).
- `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 5.3).
- `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 5.3). Commonly-used input conventions:

- *`obj.applyShear(g1, 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, 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, beta)`*
  apply magnitude (`g`) and polar angle (`beta`) 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(e, beta)`*
  apply magnitude (`e`) and polar angle (`beta`) 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.applyRotation(theta)`
  apply a rotation of `theta` (positive direction anti-clockwise) to the `GSObject`, where `theta` is an `Angle` instance (see Section 5.1).

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

- `obj.draw(..., dx=1)`
  draw an image of the `GSObject` using Discrete Fourier Transforms and interpolation to perform the image rendering. The optional image sample scale `dx` (default `dx = 1`) should use the same units as used for the `GSobject` size parameters.

- `obj.drawShoot(image, ..., dx=1)`
  draw an image of the `GSObject` by shooting a finite number of photons into a user-supplied `Image` instance, `image`, which unlike for `draw()` is a required input. The resulting rendering therefore contains stochastic noise, but uses few approximations. The optional image sample scale `dx` (default `dx = 1`) should use the same units as used for the `GSobject` size parameters.

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

```
>>> print obj.<methodName>.__doc__
```

within the Python interpreter. Alternatively follow the hyperlinks on the class names above to view the documentation based on the Python docstrings. You will see that many of the `GSObject` instances also have their own specialized methods, often for retreiving parameter values. Examples are `obj.getSigma()` for the `Gaussian`, or `obj.getHalfLightRadius()` for many of the `GSObjects`.

3. Random deviates

3.1. Random deviate classes and when to use them

Random deviates will be used when wishing to add a stochastic component to the modelling 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.

A short summary of the 8 random deviates currently implemented in GalSim, with a short description of their distributions, parameterizations and default parameter values:

- `galsim.UniformDeviate(...)`
  uniform distribution in the interval $[0, 1)$.
- `galsim.GaussianDeviate(..., mean=0., sigma=1.)`
  Gaussian distribution with `mean` *and standard deviation* `sigma`.
- `galsim.BinomialDeviate(..., N=1, p=0.5)`
  Binomial distribution for `N` *trials each of probability* `p`.
- `galsim.PoissonDeviate(..., mean=1.)`
  Poisson distribution with a single `mean` *rate.*
- `galsim.CCDNoise(..., gain=1., read_noise=0.)`
  a basic detector noise model, parameterized by `gain` *and* `read_noise`.
- `galsim.WeibullDeviate(..., a=1., b=1.)`
  Weibull distribution family (includes Rayleigh and Exponential) with shape parameters `a` *and* `b`.
- `galsim.GammaDeviate(..., alpha=1., beta=1.)`
  Gamma distribution with parameters `alpha` *and* `beta`.
- `galsim.Chi2Deviate(..., n=1.)`
  χ^2 *distribution with degrees-of-freedom parameter* `n`.

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. However, the full docstrings are available in `galsim/random.py`, so please refer there for more information, or type

```
>>> print galsim.<RandomDeviateName>.__doc__
```

within the Python interpreter.

3.2. 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 two most important and commonly-used methods for such an instance are:

- `dev.applyTo(image)`
  adds a random number, distributed according to the distribution represented by `dev`*, to each element in in a supplied* `Image` *instance* `image` *(see Section 4).*

- `dev()`
  calling the deviate directly simply returns a single new random number drawn from the distribution represented by `dev`*.*

4. Images

4.1. Image classes and when to use them

The GalSim `Image` classes store array data, along with a figure for the pixel separation in physical units and image bounds information (origin, extent). The `ImageView` provides a mutable view into `Image` instance data (although not all `Image` methods are available), and `ConstImageView` an immutable view into `Image` instance data.

If creating these objects directly you will mostly only need to use `Image` instances. The `ImageView` classes are most commonly encountered as the output of the `GSObject` instance methods `obj.draw(...)` and `obj.drawShoot(...)`. Both `Image` and `ImageView` instances can be operated on to add stochastic noise simulating real astronomical images (see Section 3), and have methods for writing to FITS format output.

There are several types of GalSim `Image`, one for each of four supported array data types:

- `galsim.ImageS(...)`;`galsim.ImageViewS(...)`;`galsim.ConstImageViewS(...)`
  for short integers (typically 16 bit).

- `galsim.ImageI(...)`;`galsim.ImageViewI(...)`;`galsim.ConstImageViewI(...)`
  for integers (typically 32 bit).

- `galsim.ImageF(...)`;`galsim.ImageViewF(...)`;`galsim.ConstImageViewF(...)`
  for single precision (typically 32 bit) floats.

- galsim.ImageD(...);galsim.ImageViewD(...);galsim.ConstImageViewD(...)
  for double precision (typically 64 bit) floats.

To access the data as a Numpy array, simply use the img.array attribute, where img is an instance of one of these Image classes. However, note that the individual elements in the array attribute are accessed as img.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

```
>>> print galsim.<ImageName>.__doc__
```

within the Python interpreter.

4.2. Important Image methods and operations

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

```
img = 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:

- img.getScale()
  get the sample scale dx for this image.
- img.setScale(dx)
  set the sample scale for this image to dx — note that this scale should use the same units adopted for the GSObject sizes.
- img.addNoise(dev)
  this adds stochastic noise, distributed as represented by the random deviate instance dev, to each element of the data array in imv. This therefore has the same effect as dev.applyTo(image) (see Section 3; also galsim/noise.py).
- img.write(fits, ...)
  write the imageView to a FITS file or object as determined by the fits input parameter (see galsim/fits.py). In Section 5.4 we discuss how to write to multi-extension FITS files.

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

```
>>> imv = img.subImage(bounds)
```

where `bounds` is a `BoundsI` instance (see Section 5.2) assigns `imv` as an `ImageView` into the sub-region of `img` lying in the area represented by `bounds`. Equivalent syntax is also

```
>>> imv = img[bounds].
```

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

```
>>> img[imv.bounds] += imv
```

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

5. Miscellaneous classes and functions

A summary of miscellaneous GalSim library objects, subcategorized into broad themes. As ever, docstrings for the *all* the classes and functions below can be accessed via

```
>>> print galsim.<Name>.__doc__
```

within the Python interpreter.

5.1. Angles

- `galsim.Angle(value, angle_unit)`
  class to represent angles and handle multiple unit types, which can be initialized very simply by a multipling a numerical value and an `AngleUnit` *instance* `angle_unit` *(see below, and* `galsim/angle.py`*).*

- `galsim.AngleUnit(radians)`
  class for holding angular unit definitions, specified on initialization in `radians`*. There are five built-in AngleUnits which are always available for use:*

  - `galsim.radians # = galsim.AngleUnit(1.)`
  - `galsim.degrees # = galsim.AngleUnit(pi / 180.)`
  - `galsim.hours # = galsim.AngleUnit(pi / 12.)`
  - `galsim.arcmin # = galsim.AngleUnit(pi / 180.  / 60.)`
  - `galsim.arcsec # = galsim.AngleUnit(pi / 180.  / 3600.)`

5.2. Bounds and Positions

- `galsim.BoundsI(...)` & `galsim.BoundsD(...)`
  classes to represent image bounds in the `x`-`y` *plane 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.*

5.3. Shear and Ellipse transformations

- `galsim.Ellipse(...)`
  class to represent ellipses and thus ellipse-type transformations. The class can be initialized using a variety of different parameter conventions (see `galsim/ellipse.py`*), including being initialized with a Shear instance (see below).*

- `galsim.Shear(...)`
  class to represent shears in a variety of ways. Like the `galsim.Ellipse`*, this class can be initialized using a variety of different parameter conventions (see* `galsim/shear.py`*). Commonly-used examples:*

  - `galsim.Shear(g1, 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, 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, beta)`
    set via magnitude (`g`*) and polar angle (*`beta`*) 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(e, beta)`
    set via magnitude (`e`*) and polar angle (*`beta`*) 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.*

5.4. Multiple image FITS output tools

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

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