Pysynphot: A Python Re-Implementation of a Legacy App in Astronomy

Victoria G. Laidler (laidler@stsci.edu) – *Computer Sciences Corporation, Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA
Perry Greenfield (perry@stsci.edu) – *Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA
Ivo Busko (busko@stsci.edu) – *Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA
Robert Jedrzejewski (rij@stsci.edu) – *Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA

Pysynphot is a package that allows astronomers to model, combine, and manipulate the spectra of stars or galaxies in simulated observations. It is being developed to replace a widely used legacy application, SYNPHOT. While retaining the data-driven philosophy of the original application, Pysynphot's architecture and improved algorithms were developed to address some of its known weaknesses. The language features available in Python and its libraries, including numpy, often enabled clean solutions to what were messy problems in the original application, and the interactive graphics capabilities of matplotlib/pylab, used with a consistent set of exposed object attributes, eliminated the need to write special-purpose plotting methods. This paper will discuss these points in some detail, as well as providing an overview of the problem domain and the object model.

Introduction

One of the things that astronomers need to do is to simulate how model stars and galaxies would look if observed through a particular telescope, camera, and filter. This is useful both for planning observations ("How long do I need to observe in order to get good signal to noise?"), and for comparing actual observations with theoretical models ("Does the real observation of this galaxy prove that my theoretical model of galaxies is correct?"). This general procedure is referred to as "synthetic photometry", because it effectively performs photometric, or brightness, measurements on synthetic (simulated) data with synthetic instruments.

This is a difficult problem to solve in generality. In addition to the intrinsic properties of a star or galaxy that determine its spectrum (the amount of light emitted as a function of wavelength), effects such as redshift and dimming by interstellar dust will also affect the spectrum when it arrives at the telescope. Real spectra are noisy with limited resolution; model spectra are smooth with potentially unlimited resolution. The response function of a telescope/instrument combination is a combination of the response of all the optical elements. And astronomers are notorious for using idiosyncratic units; in addition to the SI and cgs units, there are a variety of ways to specify flux as a function of wavelength; then there are a set of magnitude units which involve a logarithmic transformation of the flux integrated over wavelength.

A software package, SYNPHOT [Bushouse], was written in the 1980s as part of the widely-used Image Reduction and Analysis Facility, IRAF [Tody], using its proprietary language SPP. Additionally, SYNPHOT essentially has its own mini-language, in which users specify the particular combination of spectrum, bandpass, units, and functions that should be applied to construct the desired spectrum.

Motivation

As with many legacy applications, maintenance issues were a strong motivation in deciding to port to a modern language. As an old proprietary language, SPP both lacks the features of modern languages and is difficult to use with modern development tools. This raised the cost of adding new functionality; so did the rigid task-oriented architecture.

It had also become clear over the years that certain deficiencies existed in SYNPHOT at a fairly basic level:

- float arithmetic was implemented in single precision
- poor combination of wavelength tables at different resolutions
- applying redshifts sometimes lost data
- no memory caching; files used for all communications

Re-implementing SYNPHOT in Python gave us the opportunity to address these and other deficiencies.

Rather than describe Pysynphot in detail, we will provide a high-level overview of the object model, and then take a close-up look at four areas that illustrate how Python, with its object-oriented capabilities and available helper packages, made it easier for us to solve some problems, simplify others, and make a great deal of progress very quickly.

Overview of Pysynphot Objects

A Spectrum is the basic class; a Spectrum always has a wavelength table and a corresponding fluxtable in a standard internal set of units. Waveunits and Fluxunits are also associated with spectra for representation purposes. Subclasses support spectra that are created

from a file, from arrays, and from analytic functions such as a black body or a Gaussian.

A Bandpass, or SpectralElement, has a wavelength table and a corresponding dimensionless throughput table. Subclasses support bandpasses that are created from a file, from arrays, from analytic functions such as a box filter, and from a specified observing configuration of a telescope/instrument combination.

Spectrum objects can be combined with each other and with Bandpasses to produce a CompositeSpectrum.

WaveUnits and FluxUnits are created from string representations of the desired unit in the shorthand used by SYNPHOT. All the unit conversion machinery is packaged in these objects.

A Spectrum and Bandpass collaborate to produce an Observation, which is a special-purpose subclass of Spectrum used to simulate observing the Spectrum through the Bandpass.

A Spectrum also collaborates with Bandpass and Unit objects to perform renormalization ("What would this spectrum have to look like in order to have a flux of value F in units U when observed through bandpass B?").

As is evident from the above definitions, most of the objects in pysynphot have array attributes, and thus rely heavily on the array functionality provided by numpy. When Spectrum or Bandpass objects are read from or written to files, these are generally FITS files, so we also rely significantly on PyFITS. Use of these two packages is sufficiently widespread that they don't show up in any of the following closeups, but they are critical to our development effort.

There are a few other specialty classes and a number of other subclasses, but this overview is enough to illustrate the rest of the paper.

Closeup #1: Improved wavelength handling

When SYNPHOT is faced with the problem of combining spectra that are defined over different wavelength sets, it creates an array using a default grid, defined as 10000 points covering the wavelength range where the calculated passband or spectrum is non-zero. The wavelengths are spaced logarithmically over this range, such that:

log10(lambda_i) = log10(lambda_min) + (i-1)*delta

where:

- lambda_i is the ith wavelength value
- delta = (log10 (max) -log10 (min)) / (N-1)
- lambda_min = wavelength of first non-zero flux
- lambda_max = wavelength of last non-zero flux
- N = 10000

This is completely insensitive to the wavelength spacing associated with each element; the spacing in the final wavelength set is determined entirely by the total range covered. Narrow spectral features can be entirely lost when they fall entirely within one of these bins.

Pysynphot does not use a pre-determined grid like this one. Instead, a CompositeSpectrum knows about the wavelength tables of each of its components. It constructs its own wavelength table by taking the union of the points in the wavelength tables in the individual components, thus preserving the original spacing around narrow features.

Closeup #2: Units

As mentioned above, astronomers use a variety of idiosyncratic units, and need to be able to convert between them. Wavelength can be measured in microns or Angstroms (10^-8 m), or as frequency in Hz. Many of the supported fluxunits are, strictly speaking, flux densities, which represent flux per wavelength per time per area; thus the units of the flux depend on the units of the wavelength, as shown in the list below. The flux itself may be represented in photons or ergs. Magnitudes, still commonly in use by optical and infrared astronomers, are typically

$-2.5 * log_{10}(F) + ZP$

where F is the flux integrated over wavelength, and the zeropoint depends on which magnitude system you're using. To compare directly to observations, you need the flux in counts, which integrates out not only the wavelength distribution but also the effective area of the telescope's light-collecting optics.

- fnu = ergs s^-1 cm^-2 Hz^-1
- flam = ergs s^-1 cm^-2 Ang^-1
- photnu = photons s^-1 cm^-2 Hz^-1
- photlam = photons s^-1 cm^-2 Ang^-1
- jy = 10^-23 ergs s^-1 cm^-2 Hz^-1
- mjy = 10^-26 ergs s^-1 cm^-2 Hz^-1
- abmag = -2.5 log10(FNU) - 48.60
- stmag = -2.5 log10(FLAM)- 21.10
- obmag = -2.5 log10(COUNTS)
- vegamag = -2.5 log10(F/ F(VEGA))
- counts = detected counts s^-1

The SYNPHOT code includes several lengthy case statements (or the equivalent thereof), to convert from the internal units to the desired units in which a particular computation needs to be performed, and then sometimes back to the internal units.

Pysynphot's OO architecture has several benefits here. Firstly, the unit conversion code is localized within the

relevant Unit classes. Secondly, the internal representation of a spectrum always exists in the internal units (of Angstroms and Photlam), so there's never a need to convert back to it. Finally, Unit classes know whether they are flux densities, magnitudes, or counts, which simplifies the tests needed in the various conversions.

Closeup #3: From command language to OO UI

A significant portion of the SYNPHOT codebase is devoted to parsing and interpreting the mini-language in which users specify a command. These specifications can be quite long, because they can include multiple filenames of spectra to be arithmetically combined. This command language also presents a significant learning curve to new users, to whom it is not immediately obvious that the command

```
rn(z(spec(qso_template.fits),2.0),
   box(10000.0,1.0),1.00E-13,flam)
```

means

> Read a spectrum from the file qso_template.fits, then apply a redshift of z=2. Then renormalize it so that, in a 1 Angstrom box centered at 10000 Angstroms, the resulting spectrum has a flux of 1e^-13 ergs cm^-2 s^-1 A^-1.

Choosing an object-oriented user interface entirely eliminated the need for a command parser (except temporarily, for backwards compatibility). Instead of learning a command language to specify complex constructs, users work directly with the building blocks and do the construction themselves. Although this is less concise, it gives users more direct control over the process, which itself allows for easier extensibility. The learning curve is also much shallower, as the class and method names are fairly intuitive.:

```
qso=S.FileSpectrum('qso_template.fits')
qso_z=qso.redshift(2.0)
bp=S.Box(10000,1)
qso_rn=qso_z.renorm(1e-13,'flam',bp)
```

Closeup #4: Pylab gave us graphics for free

The SYNPHOT package includes several specialized tasks to provide graphics capability. These tasks have lengthy parameter lists to allow the user to specify characteristics of the plot (limits, line type, and overplotting), as well as the usual parameters with which to specify the spectrum.

The availability of matplotlib, and particularly its pylab interface, meant that we have been able to provide quite a lot of graphics capability to our users without, as yet, having to write a single line of graphics-related code. We have tuned the user interface to provide consistent attributes that are useful for plotting and annotating plots.

As the user interface develops, we will likely develop some functions to provide "standard" annotations such as labels, title, and legend. But this is functional sugar; almost all the plotting capability provided by the SYNPHOT tasks is available out of the box, and pylab provides much more versatility and control. Most of our test users are coming to pysynphot and to pylab at the same time; their reaction to the plotting capabilities has been overwhelmingly positive.

Conclusion

The need to port this legacy application became an opportunity to improve it, resulting in a re-implementation with improved architecture rather than a direct port. Python's OO features and available packages made the job much easier, and Python's ability to support functional-style programming is important in lowering the adoption barrier by astronomers.

Development and testing of pysynphot are actively ongoing, with a major milestone planned for spring 2009, when it will be used by the Exposure Time Calculators during the next observing cycle for the Hubble. We are aiming for a version 1.0 release in summer 2009.

References

[Bushouse] H. Bushouse, B. Simon, "The IRAF/STSDAS Synthetic Photometry Package", *Astronomical Data Analysis Software and Systems III*, A.S.P. Conference Series, Vol. 61, 1994

[Tody] D. Tody, "The IRAF Data Reduction and Analysis System", *Instrumentation in astronomy VI*, 1986