sncosmo Documentation

Release 1.5.3

Kyle Barbary and contributors

Nov 24, 2017

Contents

SNCosmo is a Python library for supernova cosmology analysis. It aims to make such analysis both as flexible and clear as possible.

CHAPTER 1

Installation

SNCosmo works on Python 2.7 and Python 3.4+ and requires the following Python packages:

- numpy
- scipy
- astropy
- extinction

1.1 Install using conda (recommended)

If you are using Anaconda or the conda package manager, you can install SNCosmo from the astropy channel:

```
conda install -c astropy sncosmo
```

The release of the conda version may lag behind the pip release, so you may want to check the available conda version. You can do this at the link above, or by running `conda search -c astropy sncosmo`.

1.2 Install using pip

Using pip:

```
pip install --no-deps extinction sncosmo
```

Note: The `--no-deps` flag is optional, but highly recommended if you already have numpy, scipy and astropy installed, since otherwise pip will sometimes try to "help" you by upgrading your Numpy installation, which may not always be desired.

Note: If you get a `PermissionError` this means that you do not have the required administrative access to install new packages to your Python installation. In this case you may consider using the `--user` option to install the package into your home directory. You can read more about how to do this in the pip documentation.

Do **not** install sncosmo or other third-party packages using `sudo` unless you are fully aware of the risks.

Note: You will need a C compiler (e.g. `gcc` or `clang`) to be installed for the installation to succeed.

1.3 Install latest development version

SNCosmo is being developed on github. To get the latest development version using `git`:

```
git clone git://github.com/sncosmo/sncosmo.git
cd sncosmo
```

then:

```
./setup.py install
```

As with the pip install instructions, you may want to use either `setup.py install --user` or `setup.py develop` to alter where the package is installed.

1.4 Optional dependencies

Several additional packages are recommended for enabling optional functionality in SNCosmo.

- matplotlib for plotting functions.
- iminuit for light curve fitting using the Minuit minimizer in *sncosmo.fit_lc*.
- emcee for MCMC light curve parameter estimation in *sncosmo.mcmc_lc*.
- nestle for nested sampling light curve parameter estimation in *sncosmo.nest_lc*.

iminuit, emcee and nestle can be installed using pip.

The corner package is also recommended for plotting results from the samplers *sncosmo.mcmc_lc* and *sncosmo.nest_lc*, but is not used by any part of sncosmo.

CHAPTER 2

Supernova Models

2.1 Getting Started

Create a model using the built-in "source" named `'hsiao'`:

```
>>> import sncosmo
>>> model = sncosmo.Model(source='hsiao')
```

Set the redshift, time-of-zero-phase and the amplitude:

```
>>> model.set(z=0.5, t0=55000., amplitude=1.e-10)
```

Generate synthetic photometry through an observer-frame bandpass:

```
>>> model.bandmag('desr', 'ab', [54990., 55000., 55020.])
array([ 24.82381795,  24.41496701,  25.2950865 ])
```

Equivalent values in photons / s / cm^2:

```
>>> model.bandflux('desr', [54990., 55000., 55020.])
array([  7.22413301e-05,   1.05275209e-04,   4.68034980e-05])
```

Equivalent values scaled so that 1 is equivalent to an AB magnitude of 25:

```
>>> model.bandflux('desr', [54990., 55000., 55020.], zp=25., zpsys='ab')
array([ 1.17617737,  1.71400939,  0.7620183 ])
```

Generate an observer-frame spectrum at a given time and wavelengths (in ergs/s/cm^2/Angstrom):

```
>>> model.flux(54990., [4000., 4100., 4200.])
array([  4.31210900e-20,   7.46619962e-20,   1.42182787e-19])
```

2.2 Creating a model using a built-in source

A Model in sncosmo consists of

- **One "source"** A model of the spectral evolution of the source (e.g., a supernova).
- **Zero or more "propagation effects"** Models of how intervening structures (e.g., host galaxy dust, milky way dust) affect the spectrum.

In the above example, we created a model with no propagation effects, using one of the built-in `Source` instances that sncosmo knows about: `'hsiao'`. See the full *List of Built-in Sources* that sncosmo knows about.

Note: In fact, the data for "built-in" sources are hosted remotely, downloaded as needed, and cached locally. So the first time you load a given model, you need to be connected to the internet. You will see a progress bar as the data are downloaded. By default, SNCosmo will use a subdirectory of the AstroPy cache directory for this purpose, e.g., `$HOME/.astropy/cache/sncosmo`, but this can be changed by setting the `data_dir` configuration parameter in `$HOME/.astropy/config/sncosmo.cfg`. See *Directory Configuration* for more information.

Some built-in source models have multiple versions, which can be explicitly retrieved using the *get_source* function:

```
>>> source = sncosmo.get_source('hsiao', version='2.0')
>>> model = sncosmo.Model(source=source)
```

2.3 Model parameters

Each model has a set of parameter names and values:

```
>>> model.param_names
['z', 't0', 'amplitude']
>>> model.parameters
array([ 0.,  0.,  1.])
```

These can also be retrieved as:

```
>>> model.get('z')
0.0
>>> model['z']
0.0
```

Parameter values can be set by any of the following methods:

```
>>> model.parameters[0] = 0.5
>>> model.parameters = [0.5, 0., 1.]  # set the entire array
>>> model['z'] = 0.5
>>> model.set(z=0.5)
>>> model.set(z=0.5, amplitude=2.0)  # Can specify multiple parameters
>>> model.update({'z': 0.5, 'amplitude': 2.0})
```

What do these parameters mean? The first two, `z` and `t0` are common to all *Model* instances:

- `z` is the redshift of the source.
- `t0` is the observer-frame time corresponding to the source's phase=0.

Note that in some sources phase=0 might be at explosion while others might be at max: the definition of phase is arbitrary. However, observed time is always related to phase via `time = t0 + phase * (1 + z)`

The next, `amplitude`, is specific to the particular type of source. In this case, the source is a simple spectral timeseries that can only be scaled up and down. Other sources could have other parameters that affect the shape of the spectrum at each phase.

For a given model, you can set the `amplitude` (or `x0` in case you are using a SALT model) according to a desired absolute magnitude in a specific band by using the method *`set_source_peakabsmag()`*. Note that the redshift `z` affects your result. Therefore, you could specify:

```
>>> model.set(z=1.6)
>>> model.set_source_peakabsmag(-19.0, 'bessellb', 'ab')
```

Specifically, for SALT models, it is recommended to call *`set_source_peakabsmag()`* after setting the other model parameters, such as `x1` and `c`. It probably won't make a difference if you are using the `'bessellb'` bandpass, but if you were setting the absolute magnitude in another band, it would make a small difference.

The reason for this peculiarity is that "absolute magnitude" is not a parameter in the SALT2 model, per se. The parameters are `x0`, `x1`, `c`, `t0`, and `z`. `x0` is a simple multiplicative scaling factor on the whole spectral timeseries. The `set_source_peakabsmag()` method is a convenience for setting `x0` such that the integrated flux through a given bandpass is as desired. Since the integrated flux depends on the spectral shape, it will depend on `x1` and `c`.

2.4 Creating a model with a source and effect(s)

Let's create a slightly more complex model. Again we will use the Hsiao spectral time series as a source, but this time we will add host galaxy dust.

```
>>> dust = sncosmo.CCM89Dust()
>>> model = sncosmo.Model(source='hsiao',
...                       effects=[dust],
...                       effect_names=['host'],
...                       effect_frames=['rest'])
```

The model now has additional parameters that describe the dust, `hostebv` and `hostr_v`:

```
>>> model.param_names
['z', 't0', 'amplitude', 'hostebv', 'hostr_v']
>>> model.parameters
array([ 0. ,  0. ,  1. ,  0. ,  3.1])
```

These are the parameters of the `CCM89Dust` instance we created:

```
>>> dust.param_names
['ebv', 'r_v']
```

In the model, the parameter names are prefixed with the name of the effect (`host`).

At any time you can print the model to get a nicely formatted string representation of its components and current parameter values:

```
>>> print(model)
<Model at 0x...>
source:
  class      : TimeSeriesSource
  name       : hsiao
  version    : 3.0
```

```
  phases     : [-20, .., 85] days (22 points)
  wavelengths: [1000, .., 25000] Angstroms (481 points)
effect (name='host' frame='rest'):
  class           : CCM89Dust
  wavelength range: [1250, 33333] Angstroms
parameters:
  z         = 0.0
  t0        = 0.0
  amplitude = 1.0
  hostebv   = 0.0
  hostr_v   = 3.1000000000000001
```

Also, `str(model)` will return this string rather than printing it.

2.5 Adding Milky Way dust

Dust in the Milky Way will affect the shape of an observed supernova spectrum. It is important to take this into account in our model when fitting the model to observed data. As with host galaxy dust treated above, we can model Milky Way dust as a "propagation effect". The only difference is that Milky Way dust is in the observer frame rather than the supernova rest frame. Here, we create a model with dust in *both* the SN rest frame and the observer frame:

```
>>> dust = sncosmo.CCM89Dust()
>>> model = sncosmo.Model(source='hsiao',
...                       effects=[dust, dust],
...                       effect_names=['host', 'mw'],
...                       effect_frames=['rest', 'obs'])
```

We can see that the model includes four extra parameters (two describing the host galaxy dust and two describing the milky way dust):

```
>>> model.param_names
['z', 't0', 'amplitude', 'hostebv', 'hostr_v', 'mwebv', 'mwr_v']
>>> model.parameters  # default values
array([ 0. ,  0. ,  1. ,  0. ,  3.1,  0. ,  3.1])
```

The host galaxy dust parameters are prefixed with `'host'` and the Milky Way dust parameters are prefixed with `'mw'`. These are just the names we supplied when constructing the model. The effect names have no significance beyond this. The effect frames, on the other hand, *are* significant. The only allowed values are `'rest'` (rest frame) and `'obs'` (observer frame).

A typical use pattern is to get an estimate of the amount of Milky Way dust at the location of the supernova from a dust map, and then to fix that amount of dust in the model. The following example illustrates how to do this using the Schlegel, Finkbeiner and Davis (1998) dust map with the sfdmap package. First, load the dust map (do this only once):

```
>>> import sfdmap

>>> dustmap = sfdmap.SFDMap("/path/to/dust/maps")
```

Now, for each SN you wish to fit, get the amount of dust at the SN location and set the `mwebv` model parameter appropriately. For example, if the SN is located at RA=42.8 degrees, Dec=0 degrees:

```
>>> ebv = dustmap.ebv(42.8, 0.0)

>>> model.set(mwebv=ebv)
```

```
>>> # proceed with fitting the other model parameters to the data.
```

Note that we wish to *fix* the `mwebv` model parameter rather than fitting it to the data like the other parameters: We're supposing that this value is perfectly known from the dust map. Therefore, when using a function such as *`fit_lc`* to fit the parameters, be sure *not* to include `'mwebv'` in the list of parameters to vary.

2.6 Model spectrum

To retrieve a spectrum (in ergs / s / cm^2 / Angstrom) at a given observer-frame time and set of wavelengths:

```
>>> wave = np.array([3000., 3500., 4000., 4500., 5000., 5500.])
>>> model.flux(-5., wave)
array([  5.29779465e-09,   7.77702880e-09,   7.13309678e-09,
         5.68369041e-09,   3.06860759e-09,   2.59024291e-09])
```

We can supply a list or array of times and get a 2-d array back, representing the spectrum at each time:

```
>>> model.flux([-5., 2.], wave)
array([[  5.29779465e-09,   7.77702880e-09,   7.13309678e-09,
          5.68369041e-09,   3.06860759e-09,   2.59024291e-09],
       [  2.88166481e-09,   6.15186858e-09,   7.87880448e-09,
          6.93919846e-09,   3.59077596e-09,   3.27623932e-09]])
```

Changing the model parameters changes the results:

```
>>> model.parameters
array([0., 0., 1., 0., 3.1])
>>> model.flux(-5., [4000., 4500.])
array([  7.13309678e-09,   5.68369041e-09])
>>> model.set(amplitude=2., hostebv=0.1)
>>> model.flux(-5., [4000., 4500.])
array([  9.39081327e-09,   7.86972003e-09])
```

2.7 Synthetic photometry

To integrate the spectrum through a bandpass, use the bandflux method:

```
>>> model.bandflux('sdssi', -5.)
180213.72886169454
```

Here we are using the SDSS I band, at time -5. days. The return value is in photons / s / cm^2. It is also possible to supply multiple times or bands:

```
>>> model.bandflux('sdssi', [-5., 2.])
array([ 180213.72886169,  176662.68287381])
>>> model.bandflux(['sdssi', 'sdssz'], [-5., -5.])
array([ 180213.72886169,   27697.76705621])
```

Instead of returning flux in photons / s / cm^2, the flux can be normalized to a desired zeropoint by specifying the `zp` and `zpsys` keywords, which can also be scalars, lists, or arrays.

```
>>> model.bandflux(['sdssi', 'sdssz'], [-5., -5.], zp=25., zpsys='ab')
array([  5.01036850e+09,   4.74414435e+09])
```

Instead of flux, magnitude can be returned. It works very similarly to flux:

```
>>> model.bandmag('sdssi', 'ab', [0., 1.])
array([ 22.6255077 ,  22.62566363])
>>> model.bandmag('sdssi', 'vega', [0., 1.])
array([ 22.26843273,  22.26858865])
```

We have been specifying the bandpasses as strings (`'sdssi'` and `'sdssz'`). This works because these bandpasses are in the sncosmo "registry". However, this is merely a convenience. In place of strings, we could have specified the actual *`Bandpass`* objects to which the strings correspond. See *Bandpasses* for more on how to directly create *`Bandpass`* objects.

The magnitude systems work similarly to bandpasses: `'ab'` and `'vega'` refer to built-in *`MagSystem`* objects, but you can also directly supply custom *`MagSystem`* objects. See *Magnitude Systems* for details.

2.8 Initializing Sources directly

You can initialize a source directly from your own template rather than using the built-in source templates.

2.8.1 Initializing a `TimeSeriesSource`

These sources are created directly from numpy arrays. Below, we build a very simple model, of a source with a flat spectrum at all times, rising from phase -50 to 0, then declining from phase 0 to +50.

```
>>> import numpy as np
>>> phase = np.linspace(-50., 50., 11)
>>> disp = np.linspace(3000., 8000., 6)
>>> flux = np.repeat(np.array([[0.], [1.], [2.], [3.], [4.], [5.],
...                            [4.], [3.], [2.], [1.], [0.]]),
...                  6, axis=1)
>>> source = sncosmo.TimeSeriesSource(phase, disp, flux)
```

Typically, you would then include this source in a `Model`:

```
>>> model = sncosmo.Model(source)
```

2.8.2 Initializing a `SALT2Source`

The SALT2 model is initialized directly from data files representing the model. You can initialize it by giving it a path to a directory containing the files.

```
>>> source = sncosmo.SALT2Source(modeldir='/path/to/dir')
```

By default, the initializer looks for files with names like `'salt2_template_0.dat'`, but this behavior can be altered with keyword parameters:

```
>>> source = sncosmo.SALT2Source(modeldir='/path/to/dir',
...                              m0file='mytemplate0file.dat')
```

See *`SALT2Source`* for more details.

CHAPTER 3

Bandpasses

3.1 Constructing a Bandpass

Bandpass objects represent the transmission fraction of an astronomical filter as a function of dispersion (photon wavelength, frequency or energy). They are basically simple containers for arrays of these values, with a couple special features. To get a bandpass that is in the registry (built-in):

```
>>> import sncosmo
>>> band = sncosmo.get_bandpass('sdssi')
>>> band
<Bandpass 'sdssi' at 0x...>
```

To create a Bandpass directly, you can supply arrays of wavelength and transmission values:

```
>>> wavelength = [4000., 5000.]
>>> transmission = [1., 1.]
>>> sncosmo.Bandpass(wavelength, transmission, name='tophatg')
<Bandpass 'tophatg' at 0x...>
```

By default, the first argument is assumed to be wavelength in Angstroms. To specify a different dispersion unit, use a unit from the `astropy.units` package:

```
>>> import astropy.units as u
>>> wavelength = [400., 500.]
>>> transmission = [1., 1.]
>>> Bandpass(wavelength, transmission, wave_unit=u.nm)
<Bandpass 'tophatg' at 0x...>
```

3.2 Using a Bandpass

A Bandpass acts like a continuous 1-d function, returning the transmission at supplied wavelengths (always in Angstroms):

```
>>> band([4100., 4250., 4300.])
array([ 0.,  1.,  1.])
```

Note that the transmission is zero outside the defined wavelength range. Linear interpolation is used between the defined wavelengths.

Bnadpasses have a few other useful properties. You can get the range of wavelengths where the transmission is non-zero:

```
>>> band.minwave(), band.maxwave()
(4000.0, 5000.0)
```

Or the transmission-weighted effective wavelength:

```
>>> band.wave_eff
4500.0
```

Or the name:

```
>>> band.name
'tophatg'
```

3.3 Adding Bandpasses to the Registry

You can create your own bandpasses and use them like built-ins by adding them to the registry. Suppose we want to register the 'tophatg' bandpass we created:

```
>>> sncosmo.register(band, 'tophatg')
```

Or if `band.name` has been set:

```
>>> sncosmo.register(band)  # registers band under band.name
```

After doing this, we can get the bandpass object by doing

```
>>> band = sncosmo.get_bandpass('tophatg')
```

Also, **we can pass the string `'tophatg'` to any function that takes a *Bandpass* object**. This means that you can create and register bandpasses at the top of a script, then just keep track of string identifiers throughout the rest of the script.

CHAPTER 4

Magnitude Systems

SNCosmo has facilities for converting synthetic model photometry to magnitudes in a variety of magnitude systems (or equivalently, scaling fluxes to a given zeropoint in a given magnitude system). For example, in the following code snippet, the string `'ab'` specifies that we want magnitudes on the AB magnitude system:

```
>>> model.bandmag('desr', 'ab', [54990., 55000., 55020.])
```

The string `'ab'` here refers to a built-in magnitude system (`'vega'` is another option). Behind the scenes magnitude systems are represented with *MagSystem* objects. As with *Bandpass* objects, most places in SNCosmo that require a magnitude system can take either the name of a magnitude system in the registry or an actual *MagSystem* instance. You can access these objects directly or create your own.

MagSystem objects represent the spectral flux density corresponding to magnitude zero in the given system and can be used to convert physical fluxes (in photons/s/cm^2) to magnitudes. Here's an example:

```
>>> ab = sncosmo.get_magsystem('ab')
>>> ab.zpbandflux('sdssg')
546600.83408598113
```

This example gives the number of counts (in photons) when integrating the AB spectrum (which happens to be F_nu = 3631 Jansky at all wavelengths) through the SDSS g band. This works similarly for other magnitude systems:

```
>>> vega = sncosmo.get_magsystem('vega')
>>> vega.zpbandflux('sdssg')
597541.25707788975
```

You can see that the Vega spectrum is a bit brighter than the AB spectrum in this particular bandpass. Therefore, SDSS *g* magnitudes given in Vega will be larger than if given in AB.

There are convenience methods for converting an observed flux in a bandpass to a magnitude:

```
>>> ab.band_flux_to_mag(1., 'sdssg')
14.344175725172901
>>> ab.band_mag_to_flux(14.344175725172901, 'sdssg')
0.99999999999999833
```

So, one count per second in this band is equivalent to an AB magnitude of about 14.34.

4.1 "Composite" magnitude systems

Sometimes, photometric data is reported in "magnitude systems" that don't correspond directly to any spectrophotometric standard. One example is "SDSS magnitudes" which are like AB magnitudes but with an offset in each band. These are represented in SNCosmo with the *CompositeMagSystem* class. For example:

```
>>> magsys = sncosmo.CompositeMagSystem(bands={'sdssg': ('ab', 0.01),
...                                            'sdssr': ('ab', 0.02)})
```

This defines a new magnitude system that knows about only two bandpasses. In this magnitude system, an object with magnitude zero in AB would have a magntide of 0.01 in SDSS *g* and 0.02 in SDSS *r*. Indeed, you can see that the flux corresponding to magnitude zero is slightly higher in this magnitude system than in AB:

```
>>> magsys.zpbandflux('sdssr')
502660.28545283229

>>> ab.zpbandflux('sdssr')
493485.70128115633
```

Since we've only defined the offsets for this magnitude system in a couple bands, using other bandpasses results in an error:

```
>>> magsys.zpbandflux('bessellb')
ValueError: band not defined in composite magnitude system
```

CHAPTER 5

Photometric Data

5.1 Photometric data stored in AstroPy Table

In sncosmo, photometric data for a supernova is stored in an astropy `Table`: each row in the table is a photometric observation. The table must contain certain columns. To see what such a table looks like, you can load an example with the following function:

```
>>> data = sncosmo.load_example_data()
>>> print data
     time      band        flux          fluxerr      zp  zpsys
------------- ----- ----------------- -------------- ---- -----
      55070.0 sdssg    0.813499900062 0.651728140824 25.0    ab
55072.0512821 sdssr  -0.0852238865812 0.651728140824 25.0    ab
55074.1025641 sdssi -0.00681659003089 0.651728140824 25.0    ab
55076.1538462 sdssz     2.23929135407 0.651728140824 25.0    ab
55078.2051282 sdssg  -0.0308977349373 0.651728140824 25.0    ab
55080.2564103 sdssr     2.35450321853 0.651728140824 25.0    ab
... etc ...
```

This example data table above has the minimum six columns necessary for sncosmo's light curve fitting and plotting functions to interpret the data. (There's no harm in having more columns for other supplementary data.)

Additionally, metadata about the photometric data can be stored with the table: `data.meta` is an `OrderedDict` of the metadata.

5.2 Including Covariance

If your table contains a column `'fluxcov'` (or any similar name; see below) it will be interpreted as covariance between the data points and will be used *instead* of the `'fluxerr'` column when calculating a χ^2 value in fitting functions. For each row, the `'fluxcov'` column should be a length N array, where N is the number of rows in the table. In other, words, `table['fluxcov']` should have shape `(N, N)`, where other columns like `table['time']` have shape `(N,)`.

As an example, let's add a `'fluxcov'` column to the example data table above.

```
>>> data['fluxcov'] = np.diag(data['fluxerr']**2)
>>> len(data)
40
>>> data['fluxcov'].shape
(40, 40)

# diagonal elements are error squared:
>>> data['fluxcov'][0, 0]
0.45271884317377648

>>> data['fluxerr'][0]
0.67284384754100002

# off diagonal elements are zero:
>>> data['fluxcov'][0, 1]
0.0
```

As is, this would be completely equivalent to just having the `'fluxerr'` column. But now we have the flexibility to represent non-zero off-diagonal covariance.

Note: When sub-selecting data from a table with covariance, be sure to use *sncosmo.select_data*. For example, rather than `table[mask]`, use `sncosmo.select_data(table, mask)`. This ensures that the covariance column is sliced appropriately! See the documentation for *select_data* for details.

5.3 Flexible column names

What if you'd rather call the time column `'date'`, or perhaps `'mjd'`? Good news! SNCosmo is flexible about the column names. For each column, it accepts a variety of alias names:

Column	Acceptable aliases (case-independent)	Description	Type
time	'mjd', 'mjdobs', 'date', 'time', 'jd', 'mjd_obs'	Time of observation in days	float
band	'filter', 'band', 'flt', 'bandpass'	Bandpass of observation	str
flux	'flux', 'f'	Flux of observation	float
fluxerr	'flux_error', 'fluxerr', 'fluxerror', 'flux_err', 'fe'	Gaussian uncertainty on flux	float
zp	'zp', 'zeropoint', 'zpt', 'zero_point'	Zeropoint corresponding to flux	float
zpsys	'zpsys', 'magsys', 'zpmagsys'	Magnitude system for zeropoint	str
fluxcov	'covar', 'covmat', 'covariance', 'cov'	Covariance between observations (array; optional)	ndarray

Note that each column must be present in some form or another, with no repeats. For example, you can have either a `'flux'` column or a `'f'` column, but not both.

The units of the flux and flux uncertainty are effectively given by the zeropoint system, with the zeropoint itself serving as a scaling factor: For example, if the zeropoint is `25.0` and the zeropoint system is `'vega'`, a flux of 1.0 corresponds to `10**(-25/2.5)` times the integrated flux of Vega in the given bandpass.

5.4 Reading and Writing photometric data from files

SNCosmo strives to be agnostic with respect to file format. In practice there are a plethora of different file formats, both standard and non-standard, used to represent tables. Rather than picking a single supported file format, or worse, creating yet another new "standard", we choose to leave the file format mostly up to the user: A user can use any file format as long as they can read their data into an astropy `Table`.

That said, SNCosmo does include a couple convenience functions for reading and writing tables of photometric data: *sncosmo.read_lc* and *sncosmo.write_lc*:

```
>>> data = sncosmo.load_example_data()
>>> sncosmo.write_lc(data, 'test.txt')
```

This creates an output file `test.txt` that looks like:

```
@x1 0.5
@c 0.2
@z 0.5
@x0 1.20482820761e-05
@t0 55100.0
time band flux fluxerr zp zpsys
55070.0 sdssg 0.36351153597 0.672843847541 25.0 ab
55072.0512821 sdssr -0.200801295864 0.672843847541 25.0 ab
55074.1025641 sdssi 0.307494232981 0.672843847541 25.0 ab
55076.1538462 sdssz 1.08776103656 0.672843847541 25.0 ab
55078.2051282 sdssg -0.43667895645 0.672843847541 25.0 ab
55080.2564103 sdssr 1.09780966779 0.672843847541 25.0 ab
... etc ...
```

Read the file back in:

```
>>> data2 = sncosmo.read_lc('test.txt')
```

There are a few other available formats, which can be specified using the `format` keyword:

```
>>> data = sncosmo.read_lc('test.json', format='json')
```

The supported formats are listed below. If your preferred format is not included, use a standard reader/writer from astropy or the Python universe.

Format name	Description	Notes
ascii (default)	ASCII with metadata lines marked by '@'	Not readable by standard ASCII table parsers due to metadata lines.
json	JavaScript Object Notation	Good performance, but not as human-readable as ascii
salt2	SALT2 new-style data files	
salt2-old	SALT2 old-style data files	

5.5 Manipulating data tables

Because photometric data tables are astropy Tables, they can be manipulated any way that Tables can. Here's a few things you might want to do.

Rename a column:

```
>>> data.rename_column('oldname', 'newname')
```

Add a column:

```
>>> data['zp'] = 26.
```

Add a constant value to all the entries in a given column:

```
>>> data['zp'] += 0.03
```

See the documentation on astropy tables for more information.

CHAPTER 6

Applying Cuts

It is useful to be able to apply "cuts" to data before trying to fit a model to the data. This is particularly important when using some of the "guessing" algorithms in *fit_lc* and *nest_lc* that use a minimum signal-to-noise ratio to pick "good" data points. These algorithms will raise an exception if there are no data points meeting the requirements, so it is advisable to check if the data meets the requirements beforehand.

6.1 Signal-to-noise ratio cuts

Require at least one datapoint with signal-to-noise ratio (S/N) greater than 5 (in any band):

```
>>> passes = np.max(data['flux'] / data['fluxerr']) > 5.
>>> passes
True
```

Require two bands each with at least one datapoint having S/N > 5:

```
>>> mask = data['flux'] / data['fluxerr'] > 5.
>>> passes = len(np.unique(data['band'][mask])) >= 2
>>> passes
True
```

CHAPTER 7

Simulation

First, define a set of "observations". These are the properties of our observations: the time, bandpass and depth.

```
import sncosmo
from astropy.table import Table
obs = Table({'time': [56176.19, 56188.254, 56207.172],
             'band': ['desg', 'desr', 'desi'],
             'gain': [1., 1., 1.],
             'skynoise': [191.27, 147.62, 160.40],
             'zp': [30., 30., 30.],
             'zpsys':['ab', 'ab', 'ab']})
print obs
```

```
skynoise zpsys band gain    time    zp
-------- ----- ---- ---- --------- ----
  191.27    ab desg  1.0  56176.19 30.0
  147.62    ab desr  1.0 56188.254 30.0
   160.4    ab desi  1.0 56207.172 30.0
```

Suppose we want to simulate a SN with the SALT2 model and the following parameters:

```
model = sncosmo.Model(source='salt2')
params = {'z': 0.4, 't0': 56200.0, 'x0':1.e-5, 'x1': 0.1, 'c': -0.1}
```

To get the light curve for this single SN, we'd do:

```
lcs = sncosmo.realize_lcs(obs, model, [params])
print lcs[0]
```

```
   time   band      flux        fluxerr     zp  zpsys
--------- ---- ------------- ------------- ---- -----
 56176.19 desg 96.0531272705  191.27537908 30.0    ab
56188.254 desr 456.360196623  149.22627064 30.0    ab
56207.172 desi  655.40885611 162.579572369 30.0    ab
```

Note that we've passed the function a one-element list, `[params]`, and gotten back a one-element list in return. (The `realize_lcs` function is designed to operate on lists of SNe for convenience.)

7.1 Generating SN parameters

We see above that it is straightforward to simulate SNe once we already know the parameters of each one. But what if we want to pick SN parameters from some defined distribution?

Suppose we want to generate SN parameters for all the SNe we would find in a given search area over a defined period of time. We start by defining an area and time period, as well as a maximum redshift to consider:

```
area = 1.  # area in square degrees
tmin = 56175.  # minimum time
tmax = 56225.  # maximum time
zmax = 0.7
```

First, we'd like to get the number and redshifts of all SNe that occur over our 1 square degree and 50 day time period:

```
redshifts = list(sncosmo.zdist(0., zmax, time=(tmax-tmin), area=area))
print len(redshifts), "SNe"
print "redshifts:", redshifts
```

```
9 SNe
redshifts: [0.4199710008856507, 0.3500118339133868, 0.5915676316485601, 0.
↪5857452631151785, 0.49024466410556855, 0.5732679644841575, 0.6224436826380927, 0.
↪5853477892182203, 0.5522300320124105]
```

Generate a list of SN parameters using these redshifts, drawing `x1` and `c` from normal distributions:

```
from numpy.random import uniform, normal
params = [{'x0':1.e-5, 'x1':normal(0., 1.), 'c':normal(0., 0.1),
           't0':uniform(tmin, tmax), 'z': z}
          for z in redshifts]
for p in params:
    print p
```

```
{'z': 0.4199710008856507, 'x0': 1e-05, 'x1': -0.9739877070754421, 'c': -0.
↪1465835504611458, 't0': 56191.57686616353}
{'z': 0.3500118339133868, 'x0': 1e-05, 'x1': 0.04454878604727126, 'c': -0.
↪04920811869083081, 't0': 56222.76963606611}
{'z': 0.5915676316485601, 'x0': 1e-05, 'x1': -0.26765265677262423, 'c': -0.
↪06456008680932701, 't0': 56211.706219411404}
{'z': 0.5857452631151785, 'x0': 1e-05, 'x1': 0.8255953341731204, 'c': 0.
↪08520083775049729, 't0': 56209.33583211229}
{'z': 0.49024466410556855, 'x0': 1e-05, 'x1': -0.12051827966517584, 'c': -0.
↪09490756669333822, 't0': 56189.37571007927}
{'z': 0.5732679644841575, 'x0': 1e-05, 'x1': 0.3051310078192594, 'c': -0.
↪10967604820261241, 't0': 56198.04368422346}
{'z': 0.6224436826380927, 'x0': 1e-05, 'x1': -0.6329407028587257, 'c': -0.
↪009789183239376284, 't0': 56179.88133113836}
{'z': 0.5853477892182203, 'x0': 1e-05, 'x1': 0.6373371286596669, 'c': 0.
↪05151693090038232, 't0': 56212.04579735962}
{'z': 0.5522300320124105, 'x0': 1e-05, 'x1': 0.04762095339856289, 'c': -0.
↪005018877828783951, 't0': 56182.14827040906}
```

So far so good. The only problem is that `x0` doesn't vary. We'd like it to be randomly distributed with some scatter around the Hubble line, so it should depend on the redshift. Here's an alternative:

```
params = []
for z in redshifts:
    mabs = normal(-19.3, 0.3)
    model.set(z=z)
    model.set_source_peakabsmag(mabs, 'bessellb', 'ab')
    x0 = model.get('x0')
    p = {'z':z, 't0':uniform(tmin, tmax), 'x0':x0, 'x1': normal(0., 1.), 'c':␣
↪normal(0., 0.1)}
    params.append(p)

for p in params:
    print p
```

```
{'c': -0.060104568346581566, 'x0': 2.9920355958896461e-05, 'z': 0.4199710008856507,
↪'x1': -0.677121283126299, 't0': 56217.93979718883}
{'c': 0.10405991801014292, 'x0': 2.134500759148091e-05, 'z': 0.3500118339133868, 'x1
↪': 1.6034252041294512, 't0': 56218.008314206476}
{'c': -0.14777109151711296, 'x0': 7.9108889725043354e-06, 'z': 0.5915676316485601, 'x1
↪': -2.2082282760850993, 't0': 56218.013686428785}
{'c': 0.056034777154805086, 'x0': 6.6457371815973038e-06, 'z': 0.5857452631151785, 'x1
↪': 0.675413080007434, 't0': 56189.03517395757}
{'c': -0.0709158052635228, 'x0': 1.2228145655148946e-05, 'z': 0.49024466410556855, 'x1
↪': 0.5449847454420981, 't0': 56198.02895700289}
{'c': -0.22101146234021096, 'x0': 7.4299221264917702e-06, 'z': 0.5732679644841575, 'x1
↪': -1.543245858395605, 't0': 56189.04585414441}
{'c': 0.06964843664572477, 'x0': 9.7121906557832662e-06, 'z': 0.6224436826380927, 'x1
↪': 1.7419604610283943, 't0': 56212.827270197355}
{'c': 0.07320513053870191, 'x0': 3.22205341646521e-06, 'z': 0.5853477892182203, 'x1':␣
↪-0.39658066375434153, 't0': 56200.421464066916}
{'c': 0.18555773972769227, 'x0': 7.5955258508017471e-06, 'z': 0.5522300320124105, 'x1
↪': -0.24463691193386283, 't0': 56190.492271332616}
```

Now we can generate the lightcurves for these parameters:

```
lcs = sncosmo.realize_lcs(obs, model, params)
print lcs[0]
```

```
   time   band      flux       fluxerr     zp  zpsys
--------- ---- ------------- ------------ ---- -----
 56176.19 desg 6.70520005464       191.27 30.0    ab
56188.254 desr 106.739113709       147.62 30.0    ab
56207.172 desi  1489.7521011 164.62420476 30.0    ab
```

Note that the "true" parameters are saved in the metadata of each SN:

```
lcs[0].meta
```

```
{'c': -0.060104568346581566,
 't0': 56217.93979718883,
 'x0': 2.9920355958896461e-05,
 'x1': -0.677121283126299,
 'z': 0.4199710008856507}
```

CHAPTER 8

Registry

8.1 What is it?

The registry (`sncosmo.registry`) is responsible for translating string identifiers to objects, for user convenience. For example, it is used in *`sncosmo.get_bandpass`* and *`sncosmo.get_source`* to return a *`Bandpass`* or *`sncosmo.Model`* object based on the name of the bandpass or model:

```
>>> sncosmo.get_bandpass('sdssi')
<Bandpass 'sdssi' at 0x28e7c90>
```

It is also used in methods like *`bandflux`* to give it the ability to accept either a *`Bandpass`* object or the name of a bandpass:

```
>>> model = sncosmo.Model(source='hsiao')
>>> model.bandflux('sdssg', 0.)  # works, thanks to registry.
```

Under the covers, the `bandflux` method retrieves the *`Bandpass`* corresponding to `'sdssg'` by calling the *`sncosmo.get_bandpass`* function.

The registry is actually quite simple: it basically amounts to a dictionary and a few functions for accessing the dictionary. Most of the time, a user doesn't need to know anything about the registry. However, it is useful if you want to add your own "built-ins" or change the name of existing ones.

8.2 Using the registry to achieve custom "built-ins"

There are a small set of "built-in" models, bandpasses, and magnitude systems. But what if you want additional ones?

Create a file `mydefs.py` that registers all your custom definitions:

```
# contents of mydefs.py
import numpy as np
import sncosmo
```

```
wave = np.array([4000., 4200., 4400., 4600., 4800., 5000.])
trans = np.array([0., 1., 1., 1., 1., 0.])
band = sncosmo.Bandpass(wave, trans, name='tophatg')

sncosmo.registry.register(band)
```

Make sure `mydefs.py` is somewhere in your `$PYTHONPATH` or the directory you are running your main script from. Now in your script import your definitions at the beginning:

```
>>> import sncosmo
>>> import mydefs
>>> # ... proceed as normal
>>> # you can now use 'tophatg' as a built-in
```

8.3 Changing the name of built-ins

To change the name of the `'sdssg'` band to `'SDSS_G'`:

```
# contents of mydefs.py
import sncosmo

band = sncosmo.get_bandpass('sdssg')
band.name = 'SDSS_G'
sncosmo.register(band)
```

8.4 Large built-ins

What if your built-ins are really big or you have a lot of them? You might only want to load them as they are needed, rather than having to load everything into memory when you do `import mydefs`. You can use the `sncosmo.registry.register_loader` function. Suppose we have a bandpass that requires a huge data file (In reality it is unlikely that loading bandpasses would take a noticeable amount of time, but it might for models or spectra.):

```
# contents of mydefs.py
import sncosmo

def load_bandpass(filename, name=None, version=None):
    # ...
    # read data from filename, create a Bandpass object, "band"
    # ...
    return band

filename = 'path/to/datafile/for/huge_tophatg'
sncosmo.register_loader(
    sncosmo.Bandpass,       # class of object returned.
    'huge_tophatg',         # name
    load_bandpass,          # function that does the loading
    [filename]              # arguments to pass to function
    )
```

Now when you `import mydefs` the registry will know how to load the *Bandpass* named `'huge_tophatg'` when it is needed. When loaded, it will be saved in memory so that subsequent operations don't need to load it again.

CHAPTER 9

Directory Configuration

The "built-in" Sources and Spectra in SNCosmo depend on some sizable data files. These files are hosted remotely, downloaded as needed, and cached locally. This all happens automatically, but it is helpful to know where the files are stored if you want to inspect them or share a common download directory between multiple users.

By default, SNCosmo will create and use an `sncosmo` subdirectory in the AstroPy cache directory for this purpose. For example, `$HOME/.astropy/cache/sncosmo`. After using a few models and spectra for the first time, here is what that directory might look like:

```
$ tree ~/.astropy/cache/sncosmo
/home/kyle/.astropy/cache/sncosmo
- models
|   - hsiao
|   |   - Hsiao_SED_V3.fits
|   - sako
|       - S11_SDSS-000018.SED
|       - S11_SDSS-001472.SED
|       - S11_SDSS-002000.SED
- spectra
    - alpha_lyr_stis_007.fits
    - bd_17d4708_stisnic_005.fits
```

You can see that within the top-level `$HOME/.astropy/cache/sncosmo` directory, a particular directory structure is created. This directory structure is fixed in the code, so it's best not to move things around within the top-level directory. If you do, sncosmo will think the data have not been downloaded and will re-download them.

9.1 Configuring the Directories

What if you would rather use a different directory to store downloaded data? Perhaps you'd rather the data not be in a hidden directory, or perhaps there are multiple users who wish to use a shared data directory. There are two options:

1. Set the environment variable `SNCOSMO_DATA_DIR` to the directory you wish to use. For example, in bash:

```
export SNCOSMO_DATA_DIR=/home/user/data/sncosmo
```

If this environment variable is set, it takes precedence over the second option (below).

2. Set the `data_dir` variable in the sncosmo configuartion file. This file is found in the astropy configuration directory, e.g., `$HOME/.astropy/config/sncosmo.cfg`. When you `import sncosmo` it checks for this file and creates a default one if it doesn't exist. The default one looks like this:

```
$ cat ~/.astropy/config/sncosmo.cfg

## Directory containing SFD (1998) dust maps, with names:
## 'SFD_dust_4096_ngp.fits' and 'SFD_dust_4096_sgp.fits'
## Example: sfd98_dir = /home/user/data/sfd98
# sfd98_dir = None

## Directory where sncosmo will store and read downloaded data resources.
## If None, ASTROPY_CACHE_DIR/sncosmo will be used.
## Example: data_dir = /home/user/data/sncosmo
# data_dir = None
```

To change the data directory, simply uncomment the last line and set it to the desired directory. You can even move the data directory around, as long as you update this configuration parameter accordingly.

orphan

CHAPTER 10

Examples

10.1 Fitting a light curve

This example shows how to fit the parameters of a SALT2 model to photometric light curve data.

First, we'll load an example of some photometric data.

```
from __future__ import print_function

import sncosmo

data = sncosmo.load_example_data()

print(data)
```

Out:

```
time      band       flux         fluxerr       zp  zpsys
------------- ----- --------------- -------------- ---- -----
      55070.0 sdssg   0.36351153597 0.672843847541 25.0    ab
55072.0512821 sdssr -0.200801295864 0.672843847541 25.0    ab
55074.1025641 sdssi  0.307494232981 0.672843847541 25.0    ab
55076.1538462 sdssz   1.08776103656 0.672843847541 25.0    ab
55078.2051282 sdssg  -0.43667895645 0.672843847541 25.0    ab
55080.2564103 sdssr   1.09780966779 0.672843847541 25.0    ab
55082.3076923 sdssi    3.7562685627 0.672843847541 25.0    ab
55084.3589744 sdssz   5.34858894966 0.672843847541 25.0    ab
55086.4102564 sdssg   2.82614187269 0.672843847541 25.0    ab
55088.4615385 sdssr   7.56547045054 0.672843847541 25.0    ab
          ...   ...             ...            ...  ...   ...
55129.4871795 sdssr    2.6597485586 0.672843847541 25.0    ab
55131.5384615 sdssi   3.99520404021 0.672843847541 25.0    ab
55133.5897436 sdssz   5.73989458094 0.672843847541 25.0    ab
55135.6410256 sdssg  0.330702283107 0.672843847541 25.0    ab
55137.6923077 sdssr  0.565286726579 0.672843847541 25.0    ab
```

```
55139.7435897 sdssi   3.04318346795 0.672843847541 25.0    ab
55141.7948718 sdssz   5.62692686384 0.672843847541 25.0    ab
55143.8461538 sdssg -0.722654789013 0.672843847541 25.0    ab
55145.8974359 sdssr   1.12091764262 0.672843847541 25.0    ab
55147.9487179 sdssi    2.1246695264 0.672843847541 25.0    ab
      55150.0 sdssz    5.3482175645 0.672843847541 25.0    ab
Length = 40 rows
```

An important additional note: a table of photometric data has a `band` column and a `zpsys` column that use strings to identify the bandpass (e.g., `'sdssg'`) and zeropoint system (`'ab'`) of each observation. If the bandpass and zeropoint systems in your data are *not* built-ins known to sncosmo, you must register the corresponding *Bandpass* or *MagSystem* to the right string identifier using the registry.

```
# create a model
model = sncosmo.Model(source='salt2')

# run the fit
result, fitted_model = sncosmo.fit_lc(
    data, model,
    ['z', 't0', 'x0', 'x1', 'c'],  # parameters of model to vary
    bounds={'z':(0.3, 0.7)})  # bounds on parameters (if any)
```

Out:

```
Downloading http://supernovae.in2p3.fr/salt/lib/exe/fetch.php?media=salt2_model_data-
↪2-4.tar.gz [Done]
Downloading http://sncosmo.github.io/data/bandpasses/sdss/sdss_g.dat [Done]
Downloading http://sncosmo.github.io/data/bandpasses/sdss/sdss_r.dat [Done]
Downloading http://sncosmo.github.io/data/bandpasses/sdss/sdss_i.dat [Done]
Downloading http://sncosmo.github.io/data/bandpasses/sdss/sdss_z.dat [Done]
```

The first object returned is a dictionary-like object where the keys can be accessed as attributes in addition to the typical dictionary lookup like `result['ncall']`:

```
print("Number of chi^2 function calls:", result.ncall)
print("Number of degrees of freedom in fit:", result.ndof)
print("chi^2 value at minimum:", result.chisq)
print("model parameters:", result.param_names)
print("best-fit values:", result.parameters)
print("The result contains the following attributes:\n", result.keys())
```

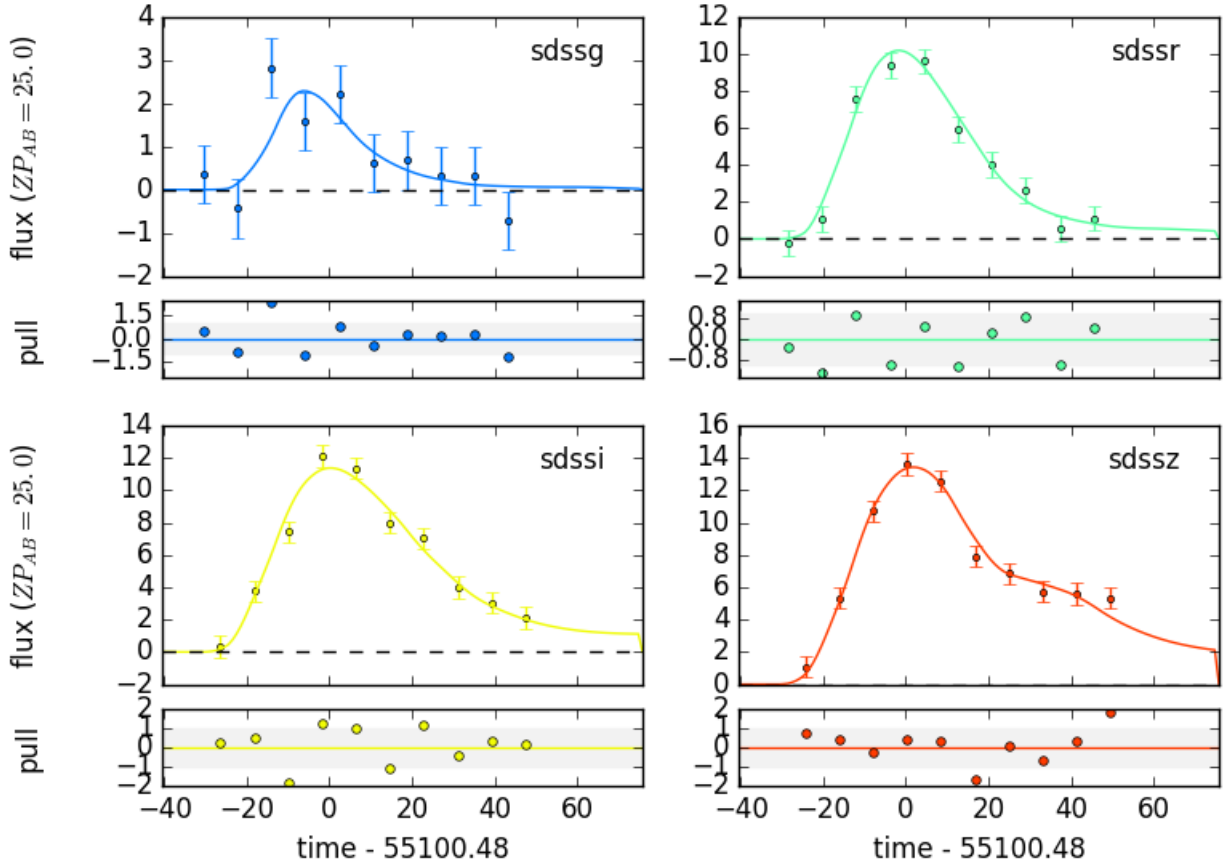
Out:

```
('Number of chi^2 function calls:', 133)
('Number of degrees of freedom in fit:', 35)
('chi^2 value at minimum:', 33.809882360763005)
('model parameters:', ['z', 't0', 'x0', 'x1', 'c'])
('best-fit values:', array([  5.15154859e-01,   5.51004778e+04,   1.19625368e-05,
         4.67270999e-01,   1.93951997e-01]))
('The result contains the following attributes:\n', ['errors', 'parameters', 'success
↪', 'data_mask', 'ndof', 'covariance', 'vparam_names', 'chisq', 'nfit', 'param_names
↪', 'message', 'ncall'])
```

The second object returned is a shallow copy of the input model with the parameters set to the best fit values. The input model is unchanged.

```
sncosmo.plot_lc(data, model=fitted_model, errors=result.errors)
```

$z = 0.515 \pm 0.015$
$t_0 = 55100.48 \pm 0.42$
$x_0 = (1.196 \pm 0.039) \times 10^{-5}$
$x_1 = 0.47 \pm 0.32$
$c = 0.194 \pm 0.037$

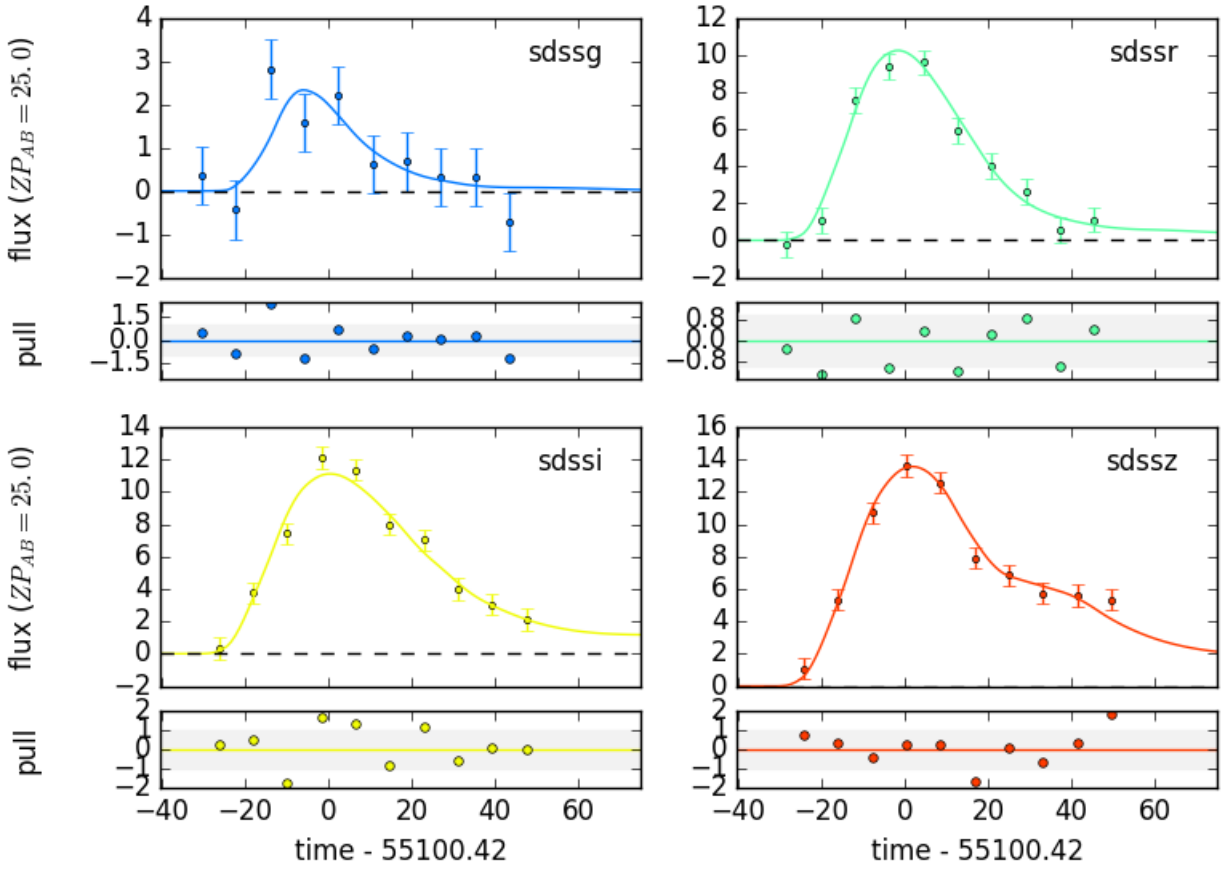


Suppose we already know the redshift of the supernova we're trying to fit. We want to set the model's redshift to the known value, and then make sure not to vary `z` in the fit.

```
model.set(z=0.5)  # set the model's redshift.
result, fitted_model = sncosmo.fit_lc(data, model,
                                      ['t0', 'x0', 'x1', 'c'])
sncosmo.plot_lc(data, model=fitted_model, errors=result.errors)
```

$z = 0.50000000$
$t_0 = 55100.42 \pm 0.41$
$x_0 = (1.181 \pm 0.038) \times 10^{-5}$

$x_1 = 0.55 \pm 0.34$
$c = 0.221 \pm 0.029$



Total running time of the script: (0 minutes 5.530 seconds)

10.2 Using a custom fitter or sampler

How to use your own minimizer or MCMC sampler for fitting light curves.

SNCosmo has three functions for model parameter estimation based on photometric data: *sncosmo.fit_lc*, *sncosmo.mcmc_lc* and *sncosmo.nest_lc*. These are wrappers around external minimizers or samplers (respectively: iminuit, emcee and nestle). However, one may wish to experiment with a custom fitting or sampling method.

Here, we give a minimal example of using the L-BFGS-B minimizer from scipy.

```
from __future__ import print_function

import numpy as np
from scipy.optimize import fmin_l_bfgs_b
import sncosmo

model = sncosmo.Model(source='salt2')
```

```
data = sncosmo.load_example_data()

# Define an objective function that we will pass to the minimizer.
# The function arguments must comply with the expectations of the specfic
# minimizer you are using.
def objective(parameters):
    model.parameters[:] = parameters  # set model parameters

    # evaluate model fluxes at times/bandpasses of data
    model_flux = model.bandflux(data['band'], data['time'],
                                zp=data['zp'], zpsys=data['zpsys'])

    # calculate and return chi^2
    return np.sum(((data['flux'] - model_flux) / data['fluxerr'])**2)

# starting parameter values in same order as `model.param_names`:
start_parameters = [0.4, 55098., 1e-5, 0., 0.]  # z, t0, x0, x1, c

# parameter bounds in same order as `model.param_names`:
bounds = [(0.3, 0.7), (55080., 55120.), (None, None), (None, None),
          (None, None)]

parameters, val, info = fmin_l_bfgs_b(objective, start_parameters,
                                      bounds=bounds, approx_grad=True)

print(parameters)
```

Out:

```
[  4.25825914e-01   5.50980000e+04   1.10729251e-05  -4.88206597e-03
   3.54030794e-01]
```

The built-in parameter estimation functions in sncosmo take care of setting up the likelihood function in the way that the underlying fitter or sampler expects. Additionally, they set guesses and bounds and package results up in a way that is as consistent as possible. For users wishing use a custom minimizer or sampler, it can be instructive to look at the source code for these functions.

Total running time of the script: (0 minutes 1.082 seconds)

10.3 Creating a new Source class

Extending sncosmo with a custom type of Source.

A `Source` is something that specifies a spectral timeseries as a function of an arbitrary number of parameters. For example, the SALT2 model has three parameters (`x0`, `x1` and `c`) that determine a unique spectrum as a function of phase. The `SALT2Source` class implements the behavior of the model: how the spectral time series depends on those parameters.

If you have a spectral timeseries model that follows the behavior of one of the existing classes, such as `TimeSeriesSource`, great! There's no need to write a custom class. However, suppose you want to implement a model that has some new parameterization. In this case, you need a new class that implements the behavior.

In this example, we implement a new type of source model. Our model is a linear combination of two spectral time series, with a parameter `w` that determines the relative weight of the models.

```
from __future__ import print_function

import numpy as np
from scipy.interpolate import RectBivariateSpline
import sncosmo


class ComboSource(sncosmo.Source):

    _param_names = ['amplitude', 'w']
    param_names_latex = ['A', 'w']   # used in plotting display

    def __init__(self, phase, wave, flux1, flux2, name=None, version=None):
        self.name = name
        self.version = version
        self._phase = phase
        self._wave = wave

        # ensure that fluxes are on the same scale
        flux2 = flux1.max() / flux2.max() * flux2

        self._model_flux1 = RectBivariateSpline(phase, wave, flux1, kx=3, ky=3)
        self._model_flux2 = RectBivariateSpline(phase, wave, flux2, kx=3, ky=3)
        self._parameters = np.array([1., 0.5])  # initial parameters

    def _flux(self, phase, wave):
        amplitude, w = self._parameters
        return amplitude * ((1.0 - w) * self._model_flux1(phase, wave) +
                            w * self._model_flux2(phase, wave))
```

... and that's all that we need to define!: A couple class attributes (`_param_names` and `param_names_latex`, an `__init__` method, and a `_flux` method. The `_flux` method is guaranteed to be passed numpy arrays for phase and wavelength.

We can now initialize an instance of this source from two spectral time series:

```
#Just as an example, we'll use some undocumented functionality in
# sncosmo to download the Nugent Ia and 2p templates. Don't rely on this
# the `DATADIR` object, or these paths in your code though, as these are
# subject to change between version of sncosmo!
from sncosmo.builtins import DATADIR
phase1, wave1, flux1 = sncosmo.read_griddata_ascii(
    DATADIR.abspath('models/nugent/sn1a_flux.v1.2.dat'))
phase2, wave2, flux2 = sncosmo.read_griddata_ascii(
    DATADIR.abspath('models/nugent/sn2p_flux.v1.2.dat'))

# In our __init__ method we defined above, the two fluxes need to be on
# the same grid, so interpolate the second onto the first:
flux2_interp = RectBivariateSpline(phase2, wave2, flux2)(phase1, wave1)

source = ComboSource(phase1, wave1, flux1, flux2_interp, name='sn1a+sn2p')
```

Out:

```
Downloading http://c3.lbl.gov/nugent/templates/sn1a_flux.v1.2.dat.gz [Done]
Downloading http://c3.lbl.gov/nugent/templates/sn2p_flux.v1.2.dat.gz [Done]
```

We can get a summary of the Source we created:

```
print(source)
```

Out:

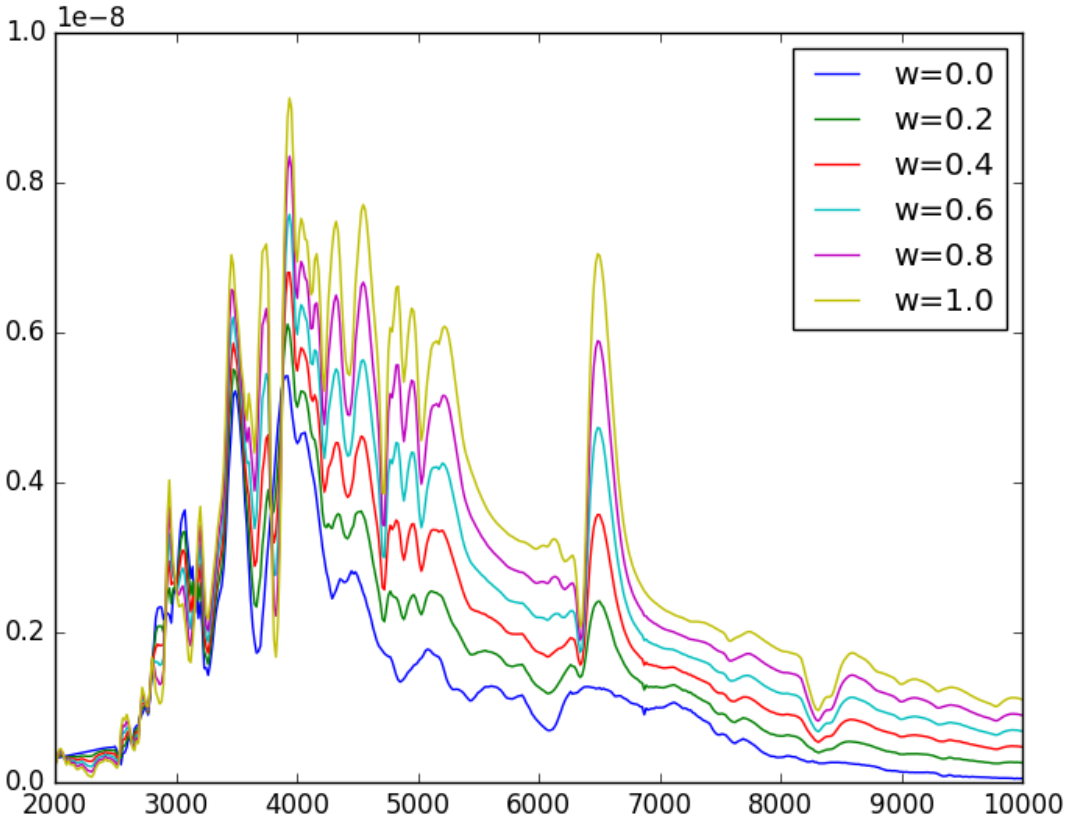
```
class      : ComboSource
name       : 'sn1a+sn2p'
version    : None
phases     : [0, .., 90] days
wavelengths: [1000, .., 25000] Angstroms
parameters:
  amplitude = 1.0
  w         = 0.5
```

Get a spectrum at phase 10 for different parameters:

```
from matplotlib import pyplot as plt

wave = np.linspace(2000.0, 10000.0, 500)
for w in (0.0, 0.2, 0.4, 0.6, 0.8, 1.0):
    source.set(w=w)
    plt.plot(wave, source.flux(10., wave), label='w={:3.1f}'.format(w))

plt.legend()
plt.show()
```



The w=0 spectrum is that of the Ia model, the w=1 spectrum is that of the IIp model, while intermediate spectra are weighted combinations.

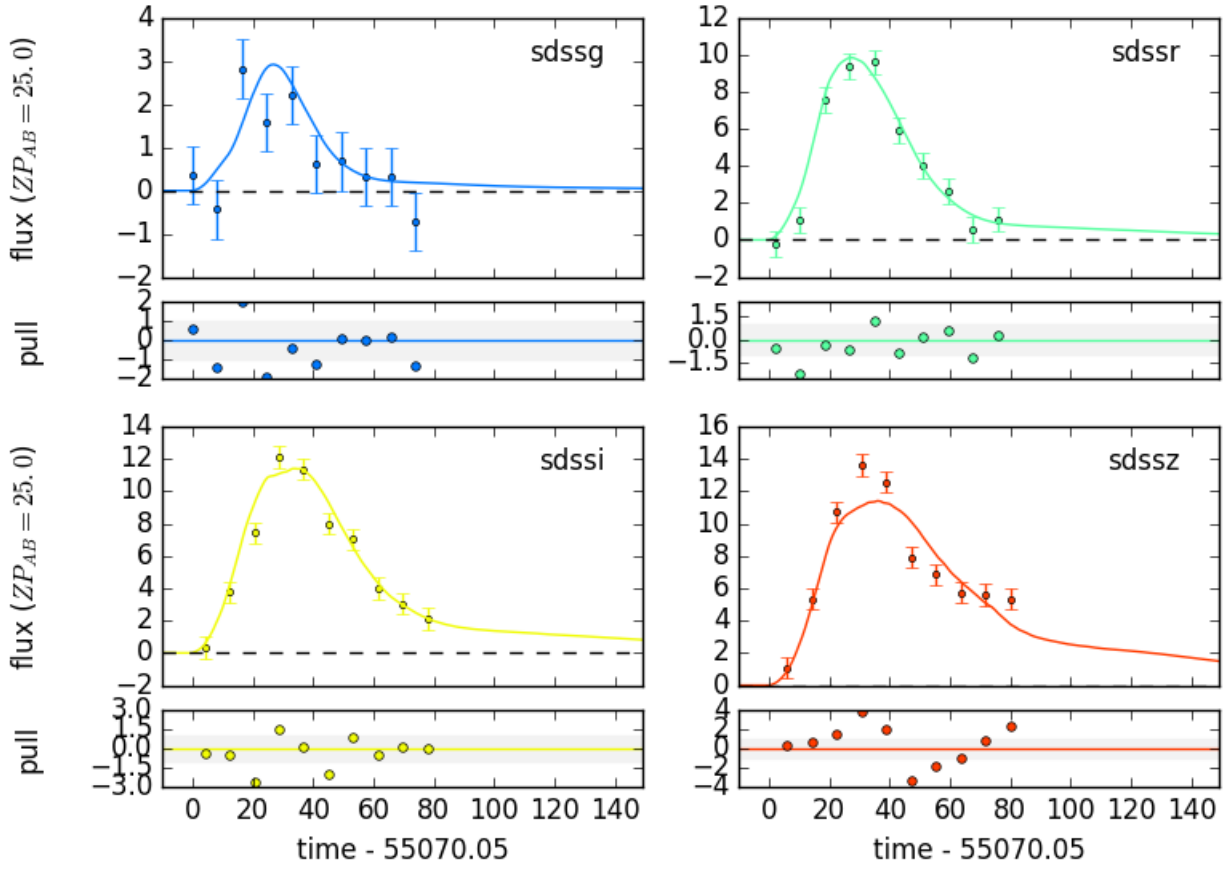
We can even fit the model to some data!

```
model = sncosmo.Model(source=source)
data = sncosmo.load_example_data()
result, fitted_model = sncosmo.fit_lc(data, model,
                                      ['z', 't0', 'amplitude', 'w'],
                                      bounds={'z': (0.2, 1.0),
                                              'w': (0.0, 1.0)})

sncosmo.plot_lc(data, model=fitted_model, errors=result.errors)
```

$z = 0.657 \pm 0.013$
$t_0 = 55070.05 \pm 0.64$

$A = (7.14 \pm 0.13) \times 10^{-10}$
$w = 0.218 \pm 0.054$



The fact that the fitted value of w is closer to 0 than 1 indicates that the light curve looks more like the Ia template than the IIp template. This is generally what we expected since the example data here was generated from a Ia template (although not the Nugent template!).

Total running time of the script: (0 minutes 4.125 seconds)

10.4 Examples

10.5 Reference / API

10.5.1 Model & Components

Model(source[, effects, effect_names, ...])	An observer-frame model, composed of a Source and zero or more effects.

sncosmo.Model

class sncosmo.**Model**(*source*, *effects=None*, *effect_names=None*, *effect_frames=None*)

An observer-frame model, composed of a Source and zero or more effects.

Parameters **source** : *Source* or str

The model for the spectral evolution of the source. If a string is given, it is used to retrieve a *Source* from the registry.

effects : list of *PropagationEffect*

List of *PropagationEffect* instances to add.

effect_names : list of str

Names of effects (same length as *effects*). The names are used to label the parameters.

effect_frames : list of str

The frame that each effect is in (same length as *effects*). Must be one of {'rest', 'obs'}.

Notes

The Source and PropagationEffects are copied upon instanciation.

Examples

```
>>> model = sncosmo.Model(source='hsiao')
```

__init__(*source*, *effects=None*, *effect_names=None*, *effect_frames=None*)

Methods

__init__(source[, effects, effect_names, ...])	
add_effect(effect, name, frame)	Add a PropagationEffect to the model.
bandflux(band, time[, zp, zpsys])	Flux through the given bandpass(es) at the given time(s).

Continued on next page

Table 10.2 – continued from previous page

bandfluxcov(band, time[, zp, zpsys])	Like bandflux(), but also returns model covariance on values.
bandmag(band, magsys, time)	Magnitude at the given time(s) through the given bandpass(es), and for the given magnitude system(s).
bandoverlap(band[, z])	Return True if model dispersion range fully overlaps the band.
color(band1, band2, magsys, time)	band1 - band2 color at the given time(s) through the given pair of
flux(time, wave)	The spectral flux density at the given time and wavelength values.
get(name)	Get parameter of the model by name.
maxtime()	Maximum observer-frame time at which the model is defined.
maxwave()	Maximum observer-frame wavelength of the model.
mintime()	Minimum observer-frame time at which the model is defined.
minwave()	Minimum observer-frame wavelength of the model.
set(**param_dict)	Set parameters of the model by name.
set_source_peakabsmag(absmag, band, magsys)	Set the amplitude of the source component of the model according to the desired absolute magnitude in the specified band.
set_source_peakmag(m, band, magsys[, sampling])	Set the amplitude of the source component of the model according to a peak apparent magnitude.
source_peakabsmag(band, magsys[, sampling, ...])	Peak absolute magnitude of the source in rest-frame bandpass.
source_peakmag(band, magsys[, sampling])	Peak apparent magnitude of source in a rest-frame bandpass.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

effect_names	Names of propagation effects (list of str).
effects	List of constituent propagation effects.
param_names	List of parameter names.
parameters	Parameter value array
source	The Source instance.

add_effect(*effect*, *name*, *frame*)
Add a PropagationEffect to the model.

Parameters **effect** : *PropagationEffect*

Propagation effect.

name : str

Name of the effect.

frame : {'rest', 'obs', 'free'}

bandflux(*band*, *time*, *zp=None*, *zpsys=None*)
Flux through the given bandpass(es) at the given time(s).

Default return value is flux in photons / s / cm^2. If zp and zpsys are given, flux(es) are scaled to the requested zeropoints.

Parameters **band** : str or list_like

Name(s) of Bandpass(es) in registry.

time : float or list_like

Time(s) in days.

zp : float or list_like, optional

If given, zeropoint to scale flux to (must also supply `zpsys`). If not given, flux is not scaled.

zpsys : str or list_like, optional

Name of a magnitude system in the registry, specifying the system that `zp` is in.

Returns **bandflux** : float or `ndarray`

Flux in photons / s /cm^2, unless `zp` and `zpsys` are given, in which case flux is scaled so that it corresponds to the requested zeropoint. Return value is `float` if all input parameters are scalars, `ndarray` otherwise.

bandfluxcov(*band*, *time*, *zp=None*, *zpsys=None*)

Like bandflux(), but also returns model covariance on values.

Parameters **band** : `bandpass` or str or list_like

Bandpass(es) or name(s) of bandpass(es) in registry.

time : float or list_like

time(s) in days.

zp : float or list_like, optional

If given, zeropoint to scale flux to. if `none` (default) flux is not scaled.

zpsys : `magsystem` or str (or list_like), optional

Determines the magnitude system of the requested zeropoint. cannot be `none` if `zp` is not `none`.

Returns **bandflux** : float or `ndarray`

Model bandfluxes.

cov : float or `array`

Covariance on `bandflux`. If `bandflux` is an array, this will be a 2-d array.

bandmag(*band*, *magsys*, *time*)

Magnitude at the given time(s) through the given bandpass(es), and for the given magnitude system(s).

Parameters **band** : str or list_like

Name(s) of bandpass in registry.

magsys : str or list_like

Name(s) of *`MagSystem`* in registry.

time : float or list_like

Observer-frame time(s) in days.

Returns **mag** : float or ndarray

Magnitude for each item in time, band, magsys. The return value is a float if all parameters are not interables. The return value is an ndarray if any are interable.

bandoverlap(*band*, *z=None*)
Return True if model dispersion range fully overlaps the band.

Parameters **band** : *Bandpass*, str or list_like

Bandpass, name of bandpass in registry, or list or array thereof.

z : float or list_like, optional

If given, evaluate the overlap when the model is at the given redshifts. If None, use the model redshift.

Returns **overlap** : bool or ndarray

color(*band1*, *band2*, *magsys*, *time*)
band1 - band2 color at the given time(s) through the given pair of bandpasses, and for the given magnitude system.

Parameters **band1** : str

Name of first bandpass in registry.

band2 : str

Name of second bandpass in registry.

magsys : str

Name of *MagSystem* in registry.

time : float or list_like

Observer-frame time(s) in days.

Returns **mag** : float or ndarray

Color for each item in time, band, magsys. The return value is a float if all parameters are not iterables. The return value is an ndarray if phase is iterable.

effect_names
Names of propagation effects (list of str).

effects
List of constituent propagation effects.

flux(*time*, *wave*)
The spectral flux density at the given time and wavelength values.

Parameters **time** : float or list_like

Time(s) in days. If None (default), the times corresponding to the native phases of the model are used.

wave : float or list_like

Wavelength(s) in Angstroms. If None (default), the native wavelengths of the model are used.

Returns **flux** : float or ndarray

Spectral flux density values in ergs / s / cm^2 / Angstrom.

get(*name*)
Get parameter of the model by name.

maxtime()
Maximum observer-frame time at which the model is defined.

maxwave()
Maximum observer-frame wavelength of the model.

mintime()
Minimum observer-frame time at which the model is defined.

minwave()
Minimum observer-frame wavelength of the model.

param_names
List of parameter names.

parameters
Parameter value array

set(***param_dict*)
Set parameters of the model by name.

set_source_peakabsmag(*absmag*, *band*, *magsys*, *sampling=1.0*, *cosmo=FlatLambdaCDM(name="WMAP9", H0=69.3 km / (Mpc s), Om0=0.286, Tcmb0=2.725 K, Neff=3.04, m_nu=[0. 0. 0.] eV, Ob0=0.0463)*)
Set the amplitude of the source component of the model according to the desired absolute magnitude in the specified band.

Parameters **absmag** : float

Desired absolute magnitude.

band : str or *Bandpass*

Bandpass or name of bandpass in registry.

magsys : str or *MagSystem*

Magnitude system or name of magnitude system in registry.

sampling : float, optional

Sampling in rest-frame days used to find the peak of the light curve. Default is 1.0.

cosmo : astropy Cosmology, optional

Instance of a cosmology from `astropy.cosmology`, used to calculate distance modulus, given the model's redshift. Default is WMAP9.

set_source_peakmag(*m*, *band*, *magsys*, *sampling=1.0*)
Set the amplitude of the source component of the model according to a peak apparent magnitude.

Note that this is the peak magnitude of just the *source* component of the model, not including effects such as dust.

Parameters **m** : float

Desired apparent magnitude.

band : str or *Bandpass*

Bandpass or name of bandpass in registry.

magsys : str or *MagSystem*

Magnitude system or name of magnitude system in registry.

sampling : float, optional

Sampling in rest-frame days used to find the peak of the light curve. Default is 1.0.

source
The Source instance.

source_peakabsmag(*band*, *magsys*, *sampling=1.0*, *cosmo=FlatLambdaCDM(name="WMAP9", H0=69.3 km / (Mpc s), Om0=0.286, Tcmb0=2.725 K, Neff=3.04, m_nu=[0. 0. 0.] eV, Ob0=0.0463)*)
Peak absolute magnitude of the source in rest-frame bandpass.

Note that this is the peak absolute magnitude of just the *source* component of the model, not including effects such as dust.

Parameters **band** : str or *Bandpass*

Bandpass or name of bandpass in registry.

magsys : str or *MagSystem*

Magnitude system or name of magnitude system in registry.

sampling : float, optional

Sampling in rest-frame days used to find the peak of the light curve. Default is 1.0.

cosmo : astropy Cosmology, optional

Instance of a cosmology from `astropy.cosmology`, used to calculate distance modulus, given the model's redshift. Default is WMAP9.

Returns float

Peak absolute magnitude of just the source component of the model.

source_peakmag(*band*, *magsys*, *sampling=1.0*)
Peak apparent magnitude of source in a rest-frame bandpass.

Note that this is the peak magnitude of just the *source* component of the model, not including effects such as dust.

Parameters **band** : str or *Bandpass*

Bandpass or name of bandpass in registry.

magsys : str or *MagSystem*

Magnitude system or name of magnitude system in registry.

sampling : float, optional

Sampling in rest-frame days used to find the peak of the light curve.

Returns float

Peak apparent magnitude of just the source component of the model.

update(*param_dict*)
Set parameters of the model from a dictionary.

Source component of Model

Source()	An abstract base class for transient models.

Continued on next page

Table 10.4 – continued from previous page

TimeSeriesSource(phase, wave, flux[, ...])	A single-component spectral time series model.
StretchSource(phase, wave, flux[, name, version])	A single-component spectral time series model, that "stretches" in time.
SALT2Source([modeldir, m0file, m1file, ...])	The SALT2 Type Ia supernova spectral timeseries model.

sncosmo.Source

class sncosmo.**Source**

An abstract base class for transient models.

A "transient model" in this case is the spectral time evolution of a source, as defined in the rest-frame of the transient: `Source` subclass instances define a spectral flux density (in, e.g., erg / s / cm^2 / Angstrom) as a function of phase and wavelength, where phase and wavelength are in the source's rest-frame. (The `Model` class wraps a `Source` instance and takes care of redshift and time dilation.) This two-dimensional spectral surface can be a function of any number of parameters that alter its amplitude or shape. Different subclasses will have different parameters.

This is an abstract base class – You can't create instances of this class. Instead, you must work with subclasses such as `TimeSeriesSource`. Subclasses must define (at minimum):

- *__init__()*
- `_param_names` (list of str)
- `_parameters` (numpy.ndarray)
- `_flux(ndarray, ndarray)`
- `minphase()`
- `maxphase()`
- `minwave()`
- `maxwave()`

__init__()

Methods

__init__()	
bandflux(band, phase[, zp, zpsys])	Flux through the given bandpass(es) at the given phase(s).
bandmag(band, magsys, phase)	Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).
flux(phase, wave)	The spectral flux density at the given phase and wavelength values.
get(name)	Get parameter of the model by name.
maxphase()	
maxwave()	
minphase()	
minwave()	
peakmag(band, magsys[, sampling])	Peak apparent magnitude in rest-frame bandpass.

Continued on next page

Table 10.5 – continued from previous page

peakphase(band_or_wave[, sampling])	Determine phase of maximum flux for the given band/wavelength.
set(**param_dict)	Set parameters of the model by name.
set_peakmag(m, band, magsys[, sampling])	Set peak apparent magnitude in rest-frame bandpass.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
parameters	Parameter value array

bandflux(*band*, *phase*, *zp=None*, *zpsys=None*)

Flux through the given bandpass(es) at the given phase(s).

Default return value is flux in photons / s / cm^2. If zp and zpsys are given, flux(es) are scaled to the requested zeropoints.

Parameters **band** : str or list_like

Name(s) of bandpass(es) in registry.

phase : float or list_like, optional

Phase(s) in days. Default is `None`, which corresponds to the full native phase sampling of the model.

zp : float or list_like, optional

If given, zeropoint to scale flux to (must also supply `zpsys`). If not given, flux is not scaled.

zpsys : str or list_like, optional

Name of a magnitude system in the registry, specifying the system that `zp` is in.

Returns **bandflux** : float or `ndarray`

Flux in photons / s /cm^2, unless `zp` and `zpsys` are given, in which case flux is scaled so that it corresponds to the requested zeropoint. Return value is `float` if all input parameters are scalars, `ndarray` otherwise.

bandmag(*band*, *magsys*, *phase*)

Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).

Parameters **band** : str or list_like

Name(s) of bandpass in registry.

magsys : str or list_like

Name(s) of *MagSystem* in registry.

phase : float or list_like

Phase(s) in days.

Returns **mag** : float or `ndarray`

Magnitude for each item in band, magsys, phase. The return value is a float if all parameters are not iterables. The return value is an `ndarray` if any are iterable.

flux (*phase*, *wave*)
The spectral flux density at the given phase and wavelength values.

Parameters **phase** : float or list_like, optional

Phase(s) in days. Must be monotonically increasing. If `None` (default), the native phases of the model are used.

wave : float or list_like, optional

Wavelength(s) in Angstroms. Must be monotonically increasing. If `None` (default), the native wavelengths of the model are used.

Returns **flux** : float or `ndarray`

Spectral flux density values in ergs / s / cm^2 / Angstrom.

get (*name*)
Get parameter of the model by name.

param_names
List of parameter names.

parameters
Parameter value array

peakmag (*band*, *magsys*, *sampling=1.0*)
Peak apparent magnitude in rest-frame bandpass.

peakphase (*band_or_wave*, *sampling=1.0*)
Determine phase of maximum flux for the given band/wavelength.

This method generates the light curve in the given band/wavelength and finds the highest-flux point. It then finds the parabola that passes through this point and the two neighboring points, and returns the position of the peak of the parabola.

set (***param_dict*)
Set parameters of the model by name.

set_peakmag (*m*, *band*, *magsys*, *sampling=1.0*)
Set peak apparent magnitude in rest-frame bandpass.

update (*param_dict*)
Set parameters of the model from a dictionary.

sncosmo.TimeSeriesSource

class sncosmo.**TimeSeriesSource** (*phase*, *wave*, *flux*, *zero_before=False*, *name=None*, *version=None*)
A single-component spectral time series model.

The spectral flux density of this model is given by

$$F(t, \lambda) = A \times M(t, \lambda)$$

where _M_ is the flux defined on a grid in phase and wavelength and _A_ (amplitude) is the single free parameter of the model. The amplitude _A_ is a simple unitless scaling factor applied to whatever flux values are used to initialize the `TimeSeriesSource`. Therefore, the _A_ parameter has no intrinsic meaning. It can only be interpreted in conjunction with the model values. Thus, it is meaningless to compare the _A_ parameter between two different `TimeSeriesSource` instances with different model data.

Parameters **phase** : `ndarray`

Phases in days.

wave : ndarray

Wavelengths in Angstroms.

flux : ndarray

Model spectral flux density in erg / s / cm^2 / Angstrom. Must have shape `(num_phases, num_wave)`.

zero_before : bool, optional

If True, flux at phases before minimum phase will be zeroed. The default is False, in which case the flux at such phases will be equal to the flux at the minimum phase (`flux[0, :]` in the input array).

name : str, optional

Name of the model. Default is `None`.

version : str, optional

Version of the model. Default is `None`.

__init__ (*phase*, *wave*, *flux*, *zero_before=False*, *name=None*, *version=None*)

Methods

__init__(phase, wave, flux[, zero_before, ...])	
bandflux(band, phase[, zp, zpsys])	Flux through the given bandpass(es) at the given phase(s).
bandmag(band, magsys, phase)	Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).
flux(phase, wave)	The spectral flux density at the given phase and wavelength values.
get(name)	Get parameter of the model by name.
maxphase()	
maxwave()	
minphase()	
minwave()	
peakmag(band, magsys[, sampling])	Peak apparent magnitude in rest-frame bandpass.
peakphase(band_or_wave[, sampling])	Determine phase of maximum flux for the given band/wavelength.
set(**param_dict)	Set parameters of the model by name.
set_peakmag(m, band, magsys[, sampling])	Set peak apparent magnitude in rest-frame bandpass.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
param_names_latex	
parameters	Parameter value array

bandflux (*band*, *phase*, *zp=None*, *zpsys=None*)

Flux through the given bandpass(es) at the given phase(s).

Default return value is flux in photons / s / cm^2. If zp and zpsys are given, flux(es) are scaled to the requested zeropoints.

Parameters **band** : str or list_like

Name(s) of bandpass(es) in registry.

phase : float or list_like, optional

Phase(s) in days. Default is `None`, which corresponds to the full native phase sampling of the model.

zp : float or list_like, optional

If given, zeropoint to scale flux to (must also supply `zpsys`). If not given, flux is not scaled.

zpsys : str or list_like, optional

Name of a magnitude system in the registry, specifying the system that `zp` is in.

Returns **bandflux** : float or `ndarray`

Flux in photons / s /cm^2, unless `zp` and `zpsys` are given, in which case flux is scaled so that it corresponds to the requested zeropoint. Return value is `float` if all input parameters are scalars, `ndarray` otherwise.

bandmag (*band*, *magsys*, *phase*)

Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).

Parameters **band** : str or list_like

Name(s) of bandpass in registry.

magsys : str or list_like

Name(s) of *`MagSystem`* in registry.

phase : float or list_like

Phase(s) in days.

Returns **mag** : float or `ndarray`

Magnitude for each item in band, magsys, phase. The return value is a float if all parameters are not iterables. The return value is an `ndarray` if any are iterable.

flux (*phase*, *wave*)

The spectral flux density at the given phase and wavelength values.

Parameters **phase** : float or list_like, optional

Phase(s) in days. Must be monotonically increasing. If `None` (default), the native phases of the model are used.

wave : float or list_like, optional

Wavelength(s) in Angstroms. Must be monotonically increasing. If `None` (default), the native wavelengths of the model are used.

Returns **flux** : float or `ndarray`

Spectral flux density values in ergs / s / cm^2 / Angstrom.

get (*name*)

Get parameter of the model by name.

param_names
List of parameter names.

parameters
Parameter value array

peakmag (*band*, *magsys*, *sampling=1.0*)
Peak apparent magnitude in rest-frame bandpass.

peakphase (*band_or_wave*, *sampling=1.0*)
Determine phase of maximum flux for the given band/wavelength.

This method generates the light curve in the given band/wavelength and finds the highest-flux point. It then finds the parabola that passes through this point and the two neighboring points, and returns the position of the peak of the parabola.

set (***param_dict*)
Set parameters of the model by name.

set_peakmag (*m*, *band*, *magsys*, *sampling=1.0*)
Set peak apparent magnitude in rest-frame bandpass.

update (*param_dict*)
Set parameters of the model from a dictionary.

sncosmo.StretchSource

class sncosmo.**StretchSource** (*phase*, *wave*, *flux*, *name=None*, *version=None*)
A single-component spectral time series model, that "stretches" in time.

The spectral flux density of this model is given by

$$F(t, \lambda) = A \times M(t/s, \lambda)$$

where _A_ is the amplitude and _s_ is the "stretch".

Parameters **phase** : ndarray

Phases in days.

wave : ndarray

Wavelengths in Angstroms.

flux : ndarray

Model spectral flux density in erg / s / cm^2 / Angstrom. Must have shape `(num_phases, num_disp)`.

__init__ (*phase*, *wave*, *flux*, *name=None*, *version=None*)

Methods

__init__(phase, wave, flux[, name, version])	
bandflux(band, phase[, zp, zpsys])	Flux through the given bandpass(es) at the given phase(s).
bandmag(band, magsys, phase)	Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).

Continued on next page

Table 10.9 – continued from previous page

flux(phase, wave)	The spectral flux density at the given phase and wavelength values.
get(name)	Get parameter of the model by name.
maxphase()	
maxwave()	
minphase()	
minwave()	
peakmag(band, magsys[, sampling])	Peak apparent magnitude in rest-frame bandpass.
peakphase(band_or_wave[, sampling])	Determine phase of maximum flux for the given band/wavelength.
set(**param_dict)	Set parameters of the model by name.
set_peakmag(m, band, magsys[, sampling])	Set peak apparent magnitude in rest-frame bandpass.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
param_names_latex	
parameters	Parameter value array

bandflux(*band*, *phase*, *zp=None*, *zpsys=None*)

Flux through the given bandpass(es) at the given phase(s).

Default return value is flux in photons / s / cm^2. If zp and zpsys are given, flux(es) are scaled to the requested zeropoints.

Parameters **band** : str or list_like

Name(s) of bandpass(es) in registry.

phase : float or list_like, optional

Phase(s) in days. Default is `None`, which corresponds to the full native phase sampling of the model.

zp : float or list_like, optional

If given, zeropoint to scale flux to (must also supply `zpsys`). If not given, flux is not scaled.

zpsys : str or list_like, optional

Name of a magnitude system in the registry, specifying the system that `zp` is in.

Returns **bandflux** : float or `ndarray`

Flux in photons / s /cm^2, unless `zp` and `zpsys` are given, in which case flux is scaled so that it corresponds to the requested zeropoint. Return value is `float` if all input parameters are scalars, `ndarray` otherwise.

bandmag(*band*, *magsys*, *phase*)

Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).

Parameters **band** : str or list_like

Name(s) of bandpass in registry.

magsys : str or list_like

Name(s) of *MagSystem* in registry.

phase : float or list_like

Phase(s) in days.

Returns **mag** : float or ndarray

Magnitude for each item in band, magsys, phase. The return value is a float if all parameters are not iterables. The return value is an ndarray if any are iterable.

flux(*phase*, *wave*)
The spectral flux density at the given phase and wavelength values.

Parameters **phase** : float or list_like, optional

Phase(s) in days. Must be monotonically increasing. If None (default), the native phases of the model are used.

wave : float or list_like, optional

Wavelength(s) in Angstroms. Must be monotonically increasing. If None (default), the native wavelengths of the model are used.

Returns **flux** : float or ndarray

Spectral flux density values in ergs / s / cm^2 / Angstrom.

get(*name*)
Get parameter of the model by name.

param_names
List of parameter names.

parameters
Parameter value array

peakmag(*band*, *magsys*, *sampling=1.0*)
Peak apparent magnitude in rest-frame bandpass.

peakphase(*band_or_wave*, *sampling=1.0*)
Determine phase of maximum flux for the given band/wavelength.

This method generates the light curve in the given band/wavelength and finds the highest-flux point. It then finds the parabola that passes through this point and the two neighboring points, and returns the position of the peak of the parabola.

set(***param_dict*)
Set parameters of the model by name.

set_peakmag(*m*, *band*, *magsys*, *sampling=1.0*)
Set peak apparent magnitude in rest-frame bandpass.

update(*param_dict*)
Set parameters of the model from a dictionary.

sncosmo.SALT2Source

class sncosmo.**SALT2Source**(*modeldir=None*, *m0file='salt2_template_0.dat'*, *m1file='salt2_template_1.dat'*, *clfile='salt2_color_correction.dat'*, *cdfile='salt2_color_dispersion.dat'*, *errscalefile='salt2_lc_dispersion_scaling.dat'*, *lcrv00file='salt2_lc_relative_variance_0.dat'*, *lcrv11file='salt2_lc_relative_variance_1.dat'*, *lcrv01file='salt2_lc_relative_covariance_01.dat'*, *name=None*, *version=None*)

The SALT2 Type Ia supernova spectral timeseries model.

The spectral flux density of this model is given by

$$F(t, \lambda) = x_0(M_0(t, \lambda) + x_1 M_1(t, \lambda)) \times 10^{-0.4CL(\lambda)c}$$

where `x0`, `x1` and `c` are the free parameters of the model, `M_0`, `M_1` are the zeroth and first components of the model, and `CL` is the colorlaw, which gives the extinction in magnitudes for `c=1`.

Parameters **modeldir** : str, optional

Directory path containing model component files. Default is `None`, which means that no directory is prepended to filenames when determining their path.

m0file, m1file, clfile : str or fileobj, optional

Filenames of various model components. Defaults are:

- m0file = 'salt2_template_0.dat' (2-d grid)
- m1file = 'salt2_template_1.dat' (2-d grid)
- clfile = 'salt2_color_correction.dat'

errscalefile, lcrv00file, lcrv11file, lcrv01file, cdfile : str or fileobj

(optional) Filenames of various model components for model covariance in synthetic photometry. See `bandflux_rcov` for details. Defaults are:

- errscalefile = 'salt2_lc_dispersion_scaling.dat' (2-d grid)
- lcrv00file = 'salt2_lc_relative_variance_0.dat' (2-d grid)
- lcrv11file = 'salt2_lc_relative_variance_1.dat' (2-d grid)
- lcrv01file = 'salt2_lc_relative_covariance_01.dat' (2-d grid)
- cdfile = 'salt2_color_dispersion.dat' (1-d grid)

Notes

The "2-d grid" files have the format `<phase> <wavelength> <value>` on each line.

The phase and wavelength values of the various components don't necessarily need to match. (In the most recent salt2 model data, they do not all match.) The phase and wavelength values of the first model component (in `m0file`) are taken as the "native" sampling of the model, even though these values might require interpolation of the other model components.

__init__(*modeldir=None*, *m0file='salt2_template_0.dat'*, *m1file='salt2_template_1.dat'*, *clfile='salt2_color_correction.dat'*, *cdfile='salt2_color_dispersion.dat'*, *errscalefile='salt2_lc_dispersion_scaling.dat'*, *lcrv00file='salt2_lc_relative_variance_0.dat'*, *lcrv11file='salt2_lc_relative_variance_1.dat'*, *lcrv01file='salt2_lc_relative_covariance_01.dat'*, *name=None*, *version=None*)

Methods

__init__([modeldir, m0file, m1file, clfile, ...])	
bandflux(band, phase[, zp, zpsys])	Flux through the given bandpass(es) at the given phase(s).
bandflux_rcov(band, phase)	Return the *relative* model covariance (or "model error") on synthetic photometry generated from the model in the given restframe band(s).
bandmag(band, magsys, phase)	Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).
colorlaw([wave])	Return the value of the CL function for the given wavelengths.
flux(phase, wave)	The spectral flux density at the given phase and wavelength values.
get(name)	Get parameter of the model by name.
maxphase()	
maxwave()	
minphase()	
minwave()	
peakmag(band, magsys[, sampling])	Peak apparent magnitude in rest-frame bandpass.
peakphase(band_or_wave[, sampling])	Determine phase of maximum flux for the given band/wavelength.
set(**param_dict)	Set parameters of the model by name.
set_peakmag(m, band, magsys[, sampling])	Set peak apparent magnitude in rest-frame bandpass.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
param_names_latex	
parameters	Parameter value array

bandflux(*band*, *phase*, *zp=None*, *zpsys=None*)

Flux through the given bandpass(es) at the given phase(s).

Default return value is flux in photons / s / cm^2. If zp and zpsys are given, flux(es) are scaled to the requested zeropoints.

Parameters **band** : str or list_like

Name(s) of bandpass(es) in registry.

phase : float or list_like, optional

Phase(s) in days. Default is `None`, which corresponds to the full native phase sampling of the model.

zp : float or list_like, optional

If given, zeropoint to scale flux to (must also supply `zpsys`). If not given, flux is not scaled.

zpsys : str or list_like, optional

Name of a magnitude system in the registry, specifying the system that `zp` is in.

Returns **bandflux** : float or `ndarray`

Flux in photons / s /cm^2, unless `zp` and `zpsys` are given, in which case flux is scaled so that it corresponds to the requested zeropoint. Return value is `float` if all input parameters are scalars, `ndarray` otherwise.

bandflux_rcov (*band*, *phase*)

Return the *relative* model covariance (or "model error") on synthetic photometry generated from the model in the given restframe band(s).

This model covariance represents the scatter of real SNe about the model. The covariance matrix has two components. The first component is diagonal (pure variance) and depends on the phase t and bandpass central wavelength λ_c of each photometry point:

$$(F_{0,\mathrm{band}}(t)/F_{1,\mathrm{band}}(t))^2 S(t,\lambda_c)^2 (V_{00}(t,\lambda_c) + 2x_1 V_{01}(t,\lambda_c) + x_1^2 V_{11}(t,\lambda_c))$$

where the 2-d functions S, V_{00}, V_{01}, and V_{11} are given by the files `errscalefile`, `lcrv00file`, `lcrv01file`, and `lcrv11file` respectively and F_0 and F_1 are given by

$$F_{0,\mathrm{band}}(t) = \int_\lambda M_0(t,\lambda) T_{\mathrm{band}}(\lambda) \frac{\lambda}{hc} d\lambda$$

$$F_{1,\mathrm{band}}(t) = \int_\lambda (M_0(t,\lambda) + x_1 M_1(t,\lambda)) T_{\mathrm{band}}(\lambda) \frac{\lambda}{hc} d\lambda$$

As this first component can sometimes be negative due to interpolation, there is a floor applied wherein values less than zero are set to `0.01**2`. This is to match the behavior of the original SALT2 code, snfit.

The second component is block diagonal. It has constant covariance between all photometry points within a bandpass (regardless of phase), and no covariance between photometry points in different bandpasses:

$$CD(\lambda_c)^2$$

where the 1-d function CD is given by the file `cdfile`. Adding these two components gives the *relative* covariance on model photometry.

Parameters **band** : `ndarray` of *Bandpass*

Bandpasses of observations.

phase : `ndarray` (float)

Phases of observations.

Returns **rcov** : `ndarray`

Model relative covariance for given bandpasses and phases.

bandmag (*band*, *magsys*, *phase*)

Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).

Parameters **band** : str or list_like

Name(s) of bandpass in registry.

magsys : str or list_like

Name(s) of *MagSystem* in registry.

phase : float or list_like

Phase(s) in days.

Returns **mag** : float or `ndarray`

Magnitude for each item in band, magsys, phase. The return value is a float if all parameters are not iterables. The return value is an `ndarray` if any are iterable.

colorlaw(*wave=None*)
Return the value of the CL function for the given wavelengths.

Parameters **wave** : float or list_like

Returns **colorlaw** : float or `ndarray`

Values of colorlaw function, which can be interpreted as extinction in magnitudes.

flux(*phase*, *wave*)
The spectral flux density at the given phase and wavelength values.

Parameters **phase** : float or list_like, optional

Phase(s) in days. Must be monotonically increasing. If `None` (default), the native phases of the model are used.

wave : float or list_like, optional

Wavelength(s) in Angstroms. Must be monotonically increasing. If `None` (default), the native wavelengths of the model are used.

Returns **flux** : float or `ndarray`

Spectral flux density values in ergs / s / cm^2 / Angstrom.

get(*name*)
Get parameter of the model by name.

param_names
List of parameter names.

parameters
Parameter value array

peakmag(*band*, *magsys*, *sampling=1.0*)
Peak apparent magnitude in rest-frame bandpass.

peakphase(*band_or_wave*, *sampling=1.0*)
Determine phase of maximum flux for the given band/wavelength.

This method generates the light curve in the given band/wavelength and finds the highest-flux point. It then finds the parabola that passes through this point and the two neighboring points, and returns the position of the peak of the parabola.

set(***param_dict*)
Set parameters of the model by name.

set_peakmag(*m*, *band*, *magsys*, *sampling=1.0*)
Set peak apparent magnitude in rest-frame bandpass.

update(*param_dict*)
Set parameters of the model from a dictionary.

Effect components of Model: interstellar dust extinction

PropagationEffect	Abstract base class for propagation effects.
CCM89Dust()	Cardelli, Clayton, Mathis (1989) extinction model dust.
OD94Dust()	O'Donnell (1994) extinction model dust.

Continued on next page

Table 10.13 – continued from previous page

F99Dust([r_v])	Fitzpatrick (1999) extinction model dust with fixed R_V.

sncosmo.PropagationEffect

class sncosmo.**PropagationEffect**

Abstract base class for propagation effects.

Derived classes must define _minwave (float), _maxwave (float).

__init__()

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

get(name)	Get parameter of the model by name.
maxwave()	
minwave()	
propagate(wave, flux)	
set(**param_dict)	Set parameters of the model by name.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
parameters	Parameter value array

get(*name*)

Get parameter of the model by name.

param_names

List of parameter names.

parameters

Parameter value array

set(**param_dict*)

Set parameters of the model by name.

update(*param_dict*)

Set parameters of the model from a dictionary.

sncosmo.CCM89Dust

class sncosmo.**CCM89Dust**

Cardelli, Clayton, Mathis (1989) extinction model dust.

__init__()

Methods

__init__()	
get(name)	Get parameter of the model by name.
maxwave()	
minwave()	
propagate(wave, flux)	Propagate the flux.
set(**param_dict)	Set parameters of the model by name.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
param_names_latex	
parameters	Parameter value array

get(*name*)
Get parameter of the model by name.

param_names
List of parameter names.

parameters
Parameter value array

propagate(*wave*, *flux*)
Propagate the flux.

set(**param_dict*)
Set parameters of the model by name.

update(*param_dict*)
Set parameters of the model from a dictionary.

sncosmo.OD94Dust

class sncosmo.**OD94Dust**
O'Donnell (1994) extinction model dust.

__init__()

Methods

__init__()	
get(name)	Get parameter of the model by name.
maxwave()	
minwave()	
propagate(wave, flux)	Propagate the flux.
set(**param_dict)	Set parameters of the model by name.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
param_names_latex	
parameters	Parameter value array

get(*name*)
Get parameter of the model by name.

param_names
List of parameter names.

parameters
Parameter value array

propagate(*wave*, *flux*)
Propagate the flux.

set(***param_dict*)
Set parameters of the model by name.

update(*param_dict*)
Set parameters of the model from a dictionary.

sncosmo.F99Dust

class sncosmo.**F99Dust**(*r_v=3.1*)
Fitzpatrick (1999) extinction model dust with fixed R_V.

__init__(*r_v=3.1*)

Methods

__init__([r_v])	
get(name)	Get parameter of the model by name.
maxwave()	
minwave()	
propagate(wave, flux)	Propagate the flux.
set(**param_dict)	Set parameters of the model by name.
update(param_dict)	Set parameters of the model from a dictionary.

Attributes

param_names	List of parameter names.
parameters	Parameter value array

get(*name*)
Get parameter of the model by name.

param_names
List of parameter names.

parameters
Parameter value array

propagate(*wave*, *flux*)
Propagate the flux.

set(***param_dict*)
Set parameters of the model by name.

update(*param_dict*)
Set parameters of the model from a dictionary.

10.5.2 Bandpass & Magnitude Systems

Bandpass(wave, trans[, wave_unit, ...])	Transmission as a function of spectral wavelength.
AggregateBandpass(transmissions[, ...])	Bandpass defined by multiple transmissions in series.
BandpassInterpolator(transmissions, ...[, ...])	Bandpass generator defined as a function of focal plane position.
MagSystem([name])	An abstract base class for magnitude systems.
ABMagSystem([name])	Magnitude system where a source with F_nu = 3631 Jansky at all frequencies has magnitude 0 in all bands.
SpectralMagSystem(refspectrum[, name])	A magnitude system defined by a fundamental spectrophotometric standard.
CompositeMagSystem([bands, families, name])	A magnitude system defined in a specific set of bands.

sncosmo.Bandpass

class sncosmo.**Bandpass**(*wave*, *trans*, *wave_unit=Unit("Angstrom")*, *trans_unit=Unit(dimensionless)*, *normalize=False*, *name=None*, *trim_level=None*)
Transmission as a function of spectral wavelength.

Parameters **wave** : list_like

Wavelength. Monotonically increasing values.

trans : list_like

Transmission fraction.

wave_unit : Unit or str, optional

Wavelength unit. Default is Angstroms.

trans_unit : Unit, optional

Transmission unit. Can be dimensionless_unscaled, indicating a ratio of transmitted to incident photons, or units proportional to inverse energy, indicating a ratio of transmitted photons to incident energy. Default is ratio of transmitted to incident photons.

normalize : bool, optional

If True, normalize fractional transmission to be 1.0 at peak. It is recommended to set normalize=True if transmission is in units of inverse energy. (When transmission is given in these units, the absolute value is usually not significant; normalizing gives more reasonable transmission values.) Default is False.

trim_level : float, optional

If given, crop bandpass to region where transmission is above this fraction of the maximum transmission. For example, if maximum transmission is 0.5, `trim_level=0.001` will remove regions where transmission is below 0.0005. Only contiguous regions on the sides of the bandpass are removed.

name : str, optional

Identifier. Default is `None`.

Examples

Construct a Bandpass and access the input arrays:

```
>>> b = Bandpass([4000., 4200., 4400.], [0.5, 1.0, 0.5])
>>> b.wave
array([ 4000.,  4200.,  4400.])
>>> b.trans
array([ 0.5,  1. ,  0.5])
```

Bandpasses act like continuous 1-d functions (linear interpolation is used):

```
>>> b([4100., 4300.])
array([ 0.75,  0.75])
```

The effective (transmission-weighted) wavelength is a property:

```
>>> b.wave_eff
4200.0
```

The `trim_level` keyword can be used to remove "out-of-band" transmission upon construction. The following example removes regions of the bandpass with tranmission less than 1 percent of peak:

```
>>> band = Bandpass([4000., 4100., 4200., 4300., 4400., 4500.],
...                 [0.001, 0.002,   0.5,   0.6, 0.003, 0.001],
...                 trim_level=0.01)
```

```
>>> band.wave
array([ 4100.,  4200.,  4300.,  4400.])
```

```
>>> band.trans
array([ 0.002,  0.5  ,  0.6  ,  0.003])
```

While less strictly correct than including the "out-of-band" transmission, only considering the region of the bandpass where transmission is significant can improve model-bandpass overlap as well as performance.

__init__(*wave*, *trans*, *wave_unit=Unit("Angstrom")*, *trans_unit=Unit(dimensionless)*, *normalize=False*, *name=None*, *trim_level=None*)

Methods

__init__(wave, trans[, wave_unit, ...])	
maxwave()	
minwave()	

Continued on next page

Table 10.23 – continued from previous page

shifted(factor[, name])	Return a new Bandpass instance with all wavelengths multiplied by a factor.
to_unit(unit)	Return wavelength and transmission in new wavelength units.

Attributes

dwave
wave_eff — Effective wavelength of bandpass in Angstroms.

shifted(*factor*, *name=None*)
Return a new Bandpass instance with all wavelengths multiplied by a factor.

to_unit(*unit*)
Return wavelength and transmission in new wavelength units.

If the requested units are the same as the current units, self is returned.

Parameters **unit** : Unit or str

Target wavelength unit.

Returns **wave** : ndarray

trans : ndarray

wave_eff
Effective wavelength of bandpass in Angstroms.

sncosmo.AggregateBandpass

class sncosmo.**AggregateBandpass**(*transmissions*, *prefactor=1.0*, *name=None*, *family=None*)
Bandpass defined by multiple transmissions in series.

Parameters **transmissions** : list of (wave, trans) pairs.

Functions defining component transmissions.

prefactor : float, optional

Scalar factor to multiply transmissions by. Default is 1.0.

name : str, optional

Name of bandpass.

family : str, optional

Name of "family" this bandpass belongs to. Such an identifier can be useful for identifying bandpasses belonging to the same instrument/filter combination but different focal plane positions.

__init__(*transmissions*, *prefactor=1.0*, *name=None*, *family=None*)

Methods

__init__(transmissions[, prefactor, name, ...])	
maxwave()	
minwave()	
shifted(factor[, name, family])	Return a new AggregateBandpass instance with all wavelengths multiplied by a factor.
to_unit(unit)	Return wavelength and transmission in new wavelength units.

Attributes

dwave	
wave_eff	Effective wavelength of bandpass in Angstroms.

shifted(*factor*, *name=None*, *family=None*)
Return a new AggregateBandpass instance with all wavelengths multiplied by a factor.

to_unit(*unit*)
Return wavelength and transmission in new wavelength units.

If the requested units are the same as the current units, self is returned.

Parameters **unit** : Unit or str

Target wavelength unit.

Returns **wave** : ndarray

trans : ndarray

wave_eff
Effective wavelength of bandpass in Angstroms.

sncosmo.BandpassInterpolator

class sncosmo.**BandpassInterpolator**(*transmissions*, *dependent_transmissions*, *prefactor=1.0*, *name=None*)
Bandpass generator defined as a function of focal plane position.

Instances of this class are not Bandpasses themselves, but generate Bandpasses at a given focal plane position. This class stores the transmission as a function of focal plane position and interpolates between the defined positions to return the bandpass at an arbitrary position.

Parameters **transmissions** : list of (wave, trans) pairs

Transmissions that apply everywhere in the focal plane.

dependent_transmissions : list of (value, wave, trans)

Transmissions that depend on some parameter. Each value is the scalar parameter value, wave and trans are 1-d arrays.

prefactor : float, optional

Scalar multiplying factor.

name : str

Examples

Transmission uniform across focal plane:

```
>>> uniform_trans = ([4000., 5000.], [1., 0.5])  # wave, trans
```

Transmissions as a function of radius:

```
>>> trans0 = (0., [4000., 5000.], [0.5, 0.5])  # radius=0
>>> trans1 = (1., [4000., 5000.], [0.75, 0.75]) # radius=1
>>> trans2 = (2., [4000., 5000.], [0.1, 0.1]) # radius=2
```

```
>>> band_interp = BandpassInterpolator([uniform_trans],
...                                    [trans0, trans1, trans2],
...                                    name='my_band')
```

Min and max radius:

```
>>> band_interp.minpos(), band_interp.maxpos()
(0.0, 2.0)
```

Get bandpass at a given radius:

```
>>> band = band_interp.at(1.5)
```

```
>>> band
<AggregateBandpass 'my_band at 1.500000' at 0x7f7a2e425668>
```

The band is aggregate of uniform transmission part, and interpolated radial-dependent part.

```
>>> band([4500., 4600.])
array([ 0.65625,  0.6125 ])
```

__init__(*transmissions*, *dependent_transmissions*, *prefactor=1.0*, *name=None*)

Methods

__init__(transmissions, dependent_transmissions)	
at(pos)	Return the bandpass at the given position
maxpos()	Maximum positional parameter value.
minpos()	Minimum positional parameter value.

at(*pos*)
Return the bandpass at the given position

maxpos()
Maximum positional parameter value.

minpos()
Minimum positional parameter value.

sncosmo.MagSystem

class sncosmo.**MagSystem**(*name=None*)
An abstract base class for magnitude systems.

__init__(*name=None*)

Methods

__init__([name])	
band_flux_to_mag(flux, band)	Convert flux (photons / s / cm^2) to magnitude.
band_mag_to_flux(mag, band)	Convert magnitude to flux in photons / s / cm^2
zpbandflux(band)	Flux of an object with magnitude zero in the given bandpass.

Attributes

name	Name of magnitude system.

band_flux_to_mag(*flux*, *band*)
Convert flux (photons / s / cm^2) to magnitude.

band_mag_to_flux(*mag*, *band*)
Convert magnitude to flux in photons / s / cm^2

name
Name of magnitude system.

zpbandflux(*band*)
Flux of an object with magnitude zero in the given bandpass.

Parameters **bandpass** : Bandpass or str

Returns **bandflux** : float

Flux in photons / s / cm^2.

sncosmo.ABMagSystem

class sncosmo.**ABMagSystem**(*name=None*)
Magnitude system where a source with F_nu = 3631 Jansky at all frequencies has magnitude 0 in all bands.

__init__(*name=None*)

Methods

__init__([name])	
band_flux_to_mag(flux, band)	Convert flux (photons / s / cm^2) to magnitude.
band_mag_to_flux(mag, band)	Convert magnitude to flux in photons / s / cm^2
zpbandflux(band)	Flux of an object with magnitude zero in the given bandpass.

Attributes

name	Name of magnitude system.

band_flux_to_mag (*flux*, *band*)
Convert flux (photons / s / cm^2) to magnitude.

band_mag_to_flux (*mag*, *band*)
Convert magnitude to flux in photons / s / cm^2

name
Name of magnitude system.

zpbandflux (*band*)
Flux of an object with magnitude zero in the given bandpass.

Parameters **bandpass** : `Bandpass` or str

Returns **bandflux** : float

Flux in photons / s / cm^2.

sncosmo.SpectralMagSystem

class `sncosmo.`**SpectralMagSystem** (*refspectrum*, *name=None*)
A magnitude system defined by a fundamental spectrophotometric standard.

Parameters **refspectrum** : `sncosmo.Spectrum`

The spectrum of the fundamental spectrophotometric standard.

__init__ (*refspectrum*, *name=None*)

Methods

__init__(refspectrum[, name])	
band_flux_to_mag(flux, band)	Convert flux (photons / s / cm^2) to magnitude.
band_mag_to_flux(mag, band)	Convert magnitude to flux in photons / s / cm^2
zpbandflux(band)	Flux of an object with magnitude zero in the given bandpass.

Attributes

name	Name of magnitude system.

band_flux_to_mag (*flux*, *band*)
Convert flux (photons / s / cm^2) to magnitude.

band_mag_to_flux (*mag*, *band*)
Convert magnitude to flux in photons / s / cm^2

name
Name of magnitude system.

zpbandflux(*band*)
Flux of an object with magnitude zero in the given bandpass.

Parameters **bandpass** : `Bandpass` or str

Returns **bandflux** : float

Flux in photons / s / cm^2.

sncosmo.CompositeMagSystem

class `sncosmo.`**CompositeMagSystem**(*bands=None*, *families=None*, *name=None*)
A magnitude system defined in a specific set of bands.

In each band, there is a fundamental standard with a known (generally non-zero) magnitude.

Parameters **bands: dict, optional**

Dictionary where keys are *Bandpass* instances or names, thereof and values are 2-tuples of magnitude system and offset. The offset gives the magnitude of standard in the given band. A positive offset means that the composite magsystem zeropoint flux is higher (brighter) than that of the standard.

families : dict, optional

Similar to the `bands` argument, but keys are strings that apply to any bandpass that has a matching `family` attribute. This is useful for generated bandpasses where the transmission differs across focal plane (and hence the bandpass at each position is unique), but all photometry has been calibrated to the same offset.

name : str

The `name` attribute of the magnitude system.

Examples

Create a magnitude system defined in only two SDSS bands where an object with AB magnitude of 0 would have a magnitude of 0.01 and 0.02 in the two bands respectively:

```
>>> sncosmo.CompositeMagSystem(bands={'sdssg': ('ab', 0.01),
...                                   'sdssr': ('ab', 0.02)})
```

__init__(*bands=None*, *families=None*, *name=None*)

Methods

__init__([bands, families, name])	
band_flux_to_mag(flux, band)	Convert flux (photons / s / cm^2) to magnitude.
band_mag_to_flux(mag, band)	Convert magnitude to flux in photons / s / cm^2
zpbandflux(band)	Flux of an object with magnitude zero in the given bandpass.

Attributes

bands	
name	Name of magnitude system.

band_flux_to_mag (*flux*, *band*)
Convert flux (photons / s / cm^2) to magnitude.

band_mag_to_flux (*mag*, *band*)
Convert magnitude to flux in photons / s / cm^2

name
Name of magnitude system.

zpbandflux (*band*)
Flux of an object with magnitude zero in the given bandpass.

Parameters **bandpass** : Bandpass or str

Returns **bandflux** : float

Flux in photons / s / cm^2.

10.5.3 I/O

Functions for reading and writing photometric data, gridded data, extinction maps, and more.

read_lc(file_or_dir[, format])	Read light curve data for a single supernova.
write_lc(data, fname[, format])	Write light curve data.
read_bandpass(fname[, fmt, wave_unit, ...])	Read bandpass from two-column ASCII file containing wavelength and transmission in each line.
load_example_data()	Load an example photometric data table.
read_snana_ascii(fname[, default_tablename])	Read an SNANA-format ascii file.
read_snana_fits(head_file, phot_file[, snids, n])	Read the SNANA FITS format: two FITS files jointly representing metadata and photometry for a set of SNe.
read_snana_simlib(fname)	Read an SNANA 'simlib' (simulation library) ascii file.
read_griddata_ascii(name_or_obj)	Read 2-d grid data from a text file.
read_griddata_fits(name_or_obj[, ext])	Read a multi-dimensional grid of data from a FITS file, where the grid coordinates are encoded in the FITS-WCS header keywords.
write_griddata_ascii(x0, x1, y, name_or_obj)	Write 2-d grid data to a text file.
write_griddata_fits(x0, x1, y, name_or_obj)	Write a 2-d grid of data to a FITS file

sncosmo.read_lc

sncosmo.**read_lc** (*file_or_dir*, *format='ascii'*, ***kwargs*)
Read light curve data for a single supernova.

Parameters **file_or_dir** : str

Filename (formats 'ascii', 'json', 'salt2') or directory name (format 'salt2-old'). For 'salt2-old' format, directory must contain a file named 'lightfile'. All other files in the directory are assumed to be photometry files, unless the filenames keyword argument is set.

format : {'ascii', 'json', 'salt2', 'salt2-old'}, optional

Format of file. Default is 'ascii'. 'salt2' is the new format available in snfit version >= 2.3.0.

read_covmat : bool, optional

[salt2 only] If True, and if a `COVMAT` keyword is present in header, read the covariance matrix from the filename specified by `COVMAT` (assumed to be in the same directory as the lightcurve file) and include it as a column named `Fluxcov` in the returned table. Default is False.

New in version 1.5.0

expand_bands : bool, optional

[salt2 only] If True, convert band names into equivalent Bandpass objects. This is particularly useful for position-dependent bandpasses in the salt2 file format: the position information is read from the header and used when creating the bandpass objects.

New in version 1.5.0

delim : str, optional

[ascii only] Used to split entries on a line. Default is `None`. Extra whitespace is ignored.

metachar : str, optional

[ascii only] Lines whose first non-whitespace character is `metachar` are treated as metadata lines, where the key and value are split on the first whitespace. Default is `'@'`

commentchar : str, optional

[ascii only] One-character string indicating a comment. Default is '#'.

filenames : list, optional

[salt2-old only] Only try to read the given filenames as photometry files. Default is to try to read all files in directory.

Returns **t** : astropy `Table`

Table of data. Metadata (as an `OrderedDict`) can be accessed via the `t.meta` attribute. For example: `t.meta['key']`. The key is case-sensitive.

Examples

Read an ascii format file that includes metadata (`StringIO` behaves like a file object):

```
>>> from astropy.extern.six import StringIO
>>> f = StringIO('''
... @id 1
... @RA 36.0
... @description good
... time band flux fluxerr zp zpsys
... 50000. g 1. 0.1 25. ab
... 50000.1 r 2. 0.1 25. ab
... ''')
>>> t = read_lc(f, format='ascii')
>>> print(t)
  time  band flux fluxerr  zp  zpsys
------- ---- ---- ------- ---- -----
50000.0    g  1.0     0.1 25.0    ab
50000.1    r  2.0     0.1 25.0    ab
```

```
>>> t.meta
OrderedDict([('id', 1), ('RA', 36.0), ('description', 'good')])
```

sncosmo.write_lc

sncosmo.**write_lc**(*data*, *fname*, *format='ascii'*, ***kwargs*)

Write light curve data.

Parameters **data** : Table

Light curve data.

fname : str

Filename.

format : {'ascii', 'salt2', 'snana', 'json'}, optional

Format of file. Default is 'ascii'. 'salt2' is the new format available in snfit version >= 2.3.0.

delim : str, optional

[ascii only] Character used to separate entries on a line. Default is ' '.

metachar : str, optional

[ascii only] Metadata designator. Default is '@'.

raw : bool, optional

[salt2, snana] By default, the SALT2 and SNANA writers rename some metadata keys and column names in order to comply with what snfit and SNANA expect. Set to True to override this. Default is False.

pedantic : bool, optional

[salt2, snana] If True, check that output column names and header keys comply with expected formatting, and raise a ValueError if not. It is probably a good idea to set to False when raw is True. Default is True.

sncosmo.read_bandpass

sncosmo.**read_bandpass**(*fname*, *fmt='ascii'*, *wave_unit=Unit("Angstrom")*, *trans_unit=Unit(dimensionless)*, *normalize=False*, *trim_level=None*, *name=None*)

Read bandpass from two-column ASCII file containing wavelength and transmission in each line.

Parameters **fname** : str

File name.

fmt : {'ascii'}

File format of file. Currently only ASCII file supported.

wave_unit : Unit or str, optional

Wavelength unit. Default is Angstroms.

trans_unit : Unit, optional

Transmission unit. Can be `dimensionless_unscaled`, indicating a ratio of transmitted to incident photons, or units proportional to inverse energy, indicating a ratio of transmitted photons to incident energy. Default is ratio of transmitted to incident photons.

normalize : bool, optional

If True, normalize fractional transmission to be 1.0 at peak. It is recommended to set to True if transmission is in units of inverse energy. (When transmission is given in these units, the absolute value is usually not significant; normalizing gives more reasonable transmission values.) Default is False.

name : str, optional

Identifier. Default is `None`.

Returns **band** : *Bandpass*

sncosmo.load_example_data

sncosmo.**load_example_data**()
Load an example photometric data table.

Returns **data** : `Table`

sncosmo.read_snana_ascii

sncosmo.**read_snana_ascii**(*fname*, *default_tablename=None*)
Read an SNANA-format ascii file.

Such files may contain metadata lines and one or more tables. See Notes for a summary of the format.

Parameters **fname** : str

Filename of object to read.

default_tablename : str, optional

Default tablename, or the string that indicates a table row, when a table starts with 'NVAR:' rather than 'NVAR_TABLENAME:'.

array : bool, optional

If True, each table is converted to a numpy array. If False, each table is a dictionary of lists (each list is a column). Default is True.

Returns **meta** : OrderedDict

Metadata from keywords.

tables : dict of `Table`

Tables, indexed by table name.

Notes

The file can contain one or more tables, as well as optional metadata. Here is an example of the expected format:

```
META1: a
META2: 6
NVAR_SN: 3
VARNAMES: A B C
SN: 1 2.0 x
SN: 4 5.0 y
```

Behavior:

- Any strings ending in a colon (:) are treated as keywords.
- The start of a new table is indicated by a keyword starting with 'NVAR'.
- If the 'NVAR' is followed by an underscore (e.g., 'NVAR_TABLENAME'), then 'TABLENAME' is taken to be the name of the table. Otherwise the user *must specify* a `default_tablename`. This is because data rows are identified by the tablename.
- After a keyword starting with 'NVAR', the next keyword must be 'VARNAMES'. The strings following give the column names.
- Any other keywords anywhere in the file are treated as metadata. The first string after the keyword is treated as the value for that keyword.
- **Note:** Newlines are treated as equivalent to spaces; they do not indicate a new row. This is necessary because some SNANA-format files have multiple metadata on a single row or single table rows split over multiple lines, making newline characters meaningless.

Examples

```
>>> from astropy.extern.six import StringIO  # StringIO behaves like a file
>>> f = StringIO('META1: a\n'
...              'META2: 6\n'
...              'NVAR_SN: 3\n'
...              'VARNAMES: A B C\n'
...              'SN: 1 2.0 x\n'
...              'SN: 4 5.0 y\n')
...
>>> meta, tables = read_snana_ascii(f)
```

The first object is a dictionary of metadata:

```
>>> meta
OrderedDict([('META1', 'a'), ('META2', 6)])
```

The second is a dictionary of all the tables in the file:

```
>>> tables['SN']
<Table rows=2 names=('A','B','C')>
array([(1, 2.0, 'x'), (4, 5.0, 'y')],
      dtype=[('A', '<i8'), ('B', '<f8'), ('C', 'S1')])
```

If the file had an 'NVAR' keyword rather than 'NVAR_SN', for example:

```
NVAR: 3
VARNAMES: A B C
SN: 1 2.0 x
SN: 4 5.0 y
SN: 5 8.2 z
```

it can be read by supplying a default table name:

```
>>> meta, tables = read_snana_ascii(f, default_tablename='SN')
```

sncosmo.read_snana_fits

sncosmo.**read_snana_fits**(*head_file*, *phot_file*, *snids=None*, *n=None*)

Read the SNANA FITS format: two FITS files jointly representing metadata and photometry for a set of SNe.

Parameters **head_file** : str

Filename of "HEAD" ("header") FITS file.

phot_file : str

Filename of "PHOT" ("photometry") FITS file.

snids : list of str, optional

If given, only return the single entry with the matching SNIDs.

n : int

If given, only return the first `n` entries.

Returns **sne** : list of `Table`

Each item in the list is an astropy Table instance.

Notes

If `head_file` contains a column 'SNID' containing strings, leading and trailing whitespace is stripped from all the values in that column.

If `phot_file` contains a column 'FLT', leading and trailing whitespace is stripped from all the values in that column.

Examples

```
>>> sne = read_snana_fits('HEAD.fits', 'PHOT.fits')
>>> for sn in sne:
...     sn.meta  # Metadata in an OrderedDict.
...     sn['MJD']  # MJD column
```

sncosmo.read_snana_simlib

sncosmo.**read_snana_simlib**(*fname*)

Read an SNANA 'simlib' (simulation library) ascii file.

Parameters **fname** : str

Filename.

Returns **meta** : `OrderedDict`

Global meta data, not associated with any one LIBID.

observation_sets : `OrderedDict` of `astropy.table.Table`

keys are LIBIDs, values are observation sets.

Notes

- Anything following '#' on each line is ignored as a comment.
- Keywords are space separated strings ending wth a colon.
- If a line starts with 'LIBID:', the following lines are associated with the value of LIBID, until 'END_LIBID:' is encountered.
- While reading a given LIBID, lines starting with 'S' or 'T' keywords are assumed to contain 12 space-separated values after the keyword. These are (1) MJD, (2) IDEXPT, (3) FLT, (4) CCD GAIN, (5) CCD NOISE, (6) SKYSIG, (7) PSF1, (8) PSF2, (9) PSF 2/1 RATIO, (10) ZPTAVG, (11) ZPTSIG, (12) MAG.
- Other lines inside a 'LIBID:'/'END_LIBID:' pair are treated as metadata for that LIBID.
- Any other keywords outside a 'LIBID:'/'END_LIBID:' pair are treated as global header keywords and are returned in the `meta` dictionary.

Examples

```
>>> meta, obs_sets = read_snana_simlib('filename')
```

The second object is a dictionary of astropy Tables indexed by LIBID:

```
>>> obs_sets.keys()
[0, 1, 2, 3, 4]
```

Each table (libid) has metadata:

```
>>> obs_sets[0].meta
OrderedDict([('LIBID', 0), ('RA', 52.5), ('DECL', -27.5), ('NOBS', 161),
             ('MWEBV', 0.0), ('PIXSIZE', 0.27)])
```

Each table has the following columns:

```
>>> obs_sets[0].colnames
['SEARCH', 'MJD', 'IDEXPT', 'FLT', 'CCD_GAIN', 'CCD_NOISE', 'SKYSIG',
 'PSF1', 'PSF2', 'PSFRATIO', 'ZPTAVG', 'ZPTSIG', 'MAG']
```

sncosmo.read_griddata_ascii

sncosmo.**read_griddata_ascii**(*name_or_obj*)

Read 2-d grid data from a text file.

Each line has values `x0 x1 y`. Space separated. x1 values are only read for first x0 value. Others are assumed to match.

Parameters **name_or_obj** : str or file-like object

Returns **x0** : numpy.ndarray

1-d array.

x1 : numpy.ndarray

1-d array.

y : numpy.ndarray

2-d array of shape (len(x0), len(x1)).

sncosmo.read_griddata_fits

sncosmo.**read_griddata_fits**(*name_or_obj*, *ext=0*)

Read a multi-dimensional grid of data from a FITS file, where the grid coordinates are encoded in the FITS-WCS header keywords.

Parameters **name_or_obj** : str or file-like object

Returns **x0, x1, ...** : `ndarray`

1-d arrays giving coordinates of grid. The number of these arrays will depend on the dimension of the data array. For example, if the data have two dimensions, a total of three arrays will be returned: `x0, x1, y`, with `x0` giving the coordinates of the first axis of `y`. If the data have three dimensions, a total of four arrays will be returned: `x0, x1, x2, y`, and so on with higher dimensions.

y : `ndarray`

n-d array of shape `(len(x0), len(x1), ...)`. For three dimensions for example, the value at `y[i, j, k]` corresponds to coordinates `(x0[i], x1[j], x2[k])`.

sncosmo.write_griddata_ascii

sncosmo.**write_griddata_ascii**(*x0*, *x1*, *y*, *name_or_obj*)

Write 2-d grid data to a text file.

Each line has values `x0 x1 y`. Space separated.

Parameters **x0** : numpy.ndarray

1-d array.

x1 : numpy.ndarray

1-d array.

y : numpy.ndarray

2-d array of shape (len(x0), len(x1)).

name_or_obj : str or file-like object

Filename to write to or open file.

sncosmo.write_griddata_fits

sncosmo.**write_griddata_fits**(*x0*, *x1*, *y*, *name_or_obj*)

Write a 2-d grid of data to a FITS file

The grid coordinates are encoded in the FITS-WCS header keywords.

Parameters **x0** : numpy.ndarray

1-d array.

x1 : numpy.ndarray

1-d array.

y : numpy.ndarray

2-d array of shape (len(x0), len(x1)).

name_or_obj : str or file-like object

Filename to write to or open file.

10.5.4 Fitting Photometric Data

Estimate model parameters from photometric data

fit_lc(data, model, vparam_names[, bounds, ...])	Fit model parameters to data by minimizing chi^2.
mcmc_lc(data, model, vparam_names[, bounds, ...])	Run an MCMC chain to get model parameter samples.
nest_lc(data, model, vparam_names, bounds[, ...])	Run nested sampling algorithm to estimate model parameters and evidence.

sncosmo.fit_lc

sncosmo.**fit_lc**(*data*, *model*, *vparam_names*, *bounds=None*, *method='minuit'*, *guess_amplitude=True*, *guess_t0=True*, *guess_z=True*, *minsnr=5.0*, *modelcov=False*, *verbose=False*, *maxcall=10000*, *phase_range=None*, *wave_range=None*, *warn=True*)

Fit model parameters to data by minimizing chi^2.

Ths function defines a chi^2 to minimize, makes initial guesses for t0 and amplitude, then runs a minimizer.

Parameters **data** : Table or ndarray or dict

Table of photometric data. Must include certain columns. See the "Photometric Data" section of the documentation for required columns.

model : *Model*

The model to fit.

vparam_names : list

Model parameters to vary in the fit.

bounds : dict, optional

Bounded range for each parameter. Keys should be parameter names, values are tuples. If a bound is not given for some parameter, the parameter is unbounded. The exception is t0: by default, the minimum bound is such that the latest phase of the model lines up with the earliest data point and the maximum bound is such that the earliest phase of the model lines up with the latest data point.

guess_amplitude : bool, optional

Whether or not to guess the amplitude from the data. If false, the current model amplitude is taken as the initial value. Only has an effect when fitting amplitude. Default is True.

guess_t0 : bool, optional

Whether or not to guess t0. Only has an effect when fitting t0. Default is True.

guess_z : bool, optional

Whether or not to guess z (redshift). Only has an effect when fitting redshift. Default is True.

minsnr : float, optional

When guessing amplitude and t0, only use data with signal-to-noise ratio (flux / fluxerr) greater than this value. Default is 5.

method : {'minuit'}, optional

Minimization method to use. Currently there is only one choice.

modelcov : bool, optional

Include model covariance when calculating chisq. Default is False. If true, the fit is performed multiple times until convergence.

phase_range : (float, float), optional

If given, discard data outside this range of phases. Note that **the definition of phase varies between models**: For example, phase=0 refers to explosion time in some models and time of peak B band flux in others.

New in version 1.5.0

wave_range : (float, float), optional

If given, discard data with bandpass effective wavelengths outside this range.

New in version 1.5.0

verbose : bool, optional

Print messages during fitting.

warn : bool, optional

Issue a warning when dropping bands outside the wavelength range of the model. Default is True.

New in version 1.5.0

Returns **res** : Result

The optimization result represented as a `Result` object, which is a `dict` subclass with attribute access. Therefore, `res.keys()` provides a list of the attributes. Attributes are:

- `success`: boolean describing whether fit succeeded.
- `message`: string with more information about exit status.
- `ncall`: number of function evaluations.
- `chisq`: minimum chi^2 value.
- `ndof`: number of degrees of freedom (len(data) - len(vparam_names)).
- `param_names`: same as `model.param_names`.
- `parameters`: 1-d `ndarray` of best-fit values (including fixed parameters) corresponding to `param_names`.
- `vparam_names`: list of varied parameter names.
- `covariance`: 2-d `ndarray` of parameter covariance; indicies correspond to order of `vparam_names`.

- `errors`: OrderedDict of varied parameter uncertainties. Corresponds to square root of diagonal entries in covariance matrix.
- `nfit`: number of times the fit was performed. Can be greater than one when model covariance, phase range or wavelength range is used. *New in version 1.5.0.*
- `data_mask`: Boolean array the same length as data specifying whether each observation was used in the final fit. *New in version 1.5.0.*

fitmodel : *Model*

A copy of the model with parameters set to best-fit values.

Notes

t0 guess: If `t0` is being fit and `guess_t0=True`, the function will guess the initial starting point for `t0` based on the data. The guess is made as follows:

- Evaluate the time and value of peak flux for the model in each band given the current model parameters.
- Determine the data point with maximum flux in each band, for points with signal-to-noise ratio > `minsnr` (default is 5). If no points meet this criteria, the band is ignored (for the purpose of guessing only).
- For each band, compare model's peak flux to the peak data point. Choose the band with the highest ratio of data / model.
- Set `t0` so that the model's time of peak in the chosen band corresponds to the peak data point in this band.

amplitude guess: If amplitude (assumed to be the first model parameter) is being fit and `guess_amplitude=True`, the function will guess the initial starting point for the amplitude based on the data.

redshift guess: If redshift (`z`) is being fit and `guess_z=True`, the function will set the initial value of `z` to the average of the bounds on `z`.

Examples

The *flatten_result* function can be used to make the result a dictionary suitable for appending as rows of a table:

```
>>> from astropy.table import Table
>>> table_rows = []
>>> for sn in sne:
...     res, fitmodel = sncosmo.fit_lc(
...          sn, model, ['t0', 'x0', 'x1', 'c'])
...     table_rows.append(flatten_result(res))
>>> t = Table(table_rows)
```

sncosmo.mcmc_lc

sncosmo.**mcmc_lc**(*data*, *model*, *vparam_names*, *bounds=None*, *priors=None*, *guess_amplitude=True*, *guess_t0=True*, *guess_z=True*, *minsnr=5.0*, *modelcov=False*, *nwalkers=10*, *nburn=200*, *nsamples=1000*, *sampler='ensemble'*, *ntemps=4*, *thin=1*, *a=2.0*, *warn=True*)

Run an MCMC chain to get model parameter samples.

This is a convenience function around `emcee.EnsembleSampler` andx `emcee.PTSampler`. It defines the likelihood function and makes a heuristic guess at a good set of starting points for the walkers. It then runs the sampler, starting with a burn-in run.

If you're not getting good results, you might want to try increasing the burn-in, increasing the walkers, or specifying a better starting position. To get a better starting position, you could first run *fit_lc*, then run this function with all `guess_[name]` keyword arguments set to False, so that the current model parameters are used as the starting point.

Parameters **data** : `Table` or `ndarray` or `dict`

Table of photometric data. Must include certain columns. See the "Photometric Data" section of the documentation for required columns.

model : *Model*

The model to fit.

vparam_names : iterable

Model parameters to vary.

bounds : `dict`, optional

Bounded range for each parameter. Keys should be parameter names, values are tuples. If a bound is not given for some parameter, the parameter is unbounded. The exception is `t0`: by default, the minimum bound is such that the latest phase of the model lines up with the earliest data point and the maximum bound is such that the earliest phase of the model lines up with the latest data point.

priors : `dict`, optional

Prior probability functions. Keys are parameter names, values are functions that return probability given the parameter value. The default prior is a flat distribution.

guess_amplitude : bool, optional

Whether or not to guess the amplitude from the data. If false, the current model amplitude is taken as the initial value. Only has an effect when fitting amplitude. Default is True.

guess_t0 : bool, optional

Whether or not to guess t0. Only has an effect when fitting t0. Default is True.

guess_z : bool, optional

Whether or not to guess z (redshift). Only has an effect when fitting redshift. Default is True.

minsnr : float, optional

When guessing amplitude and t0, only use data with signal-to-noise ratio (flux / fluxerr) greater than this value. Default is 5.

modelcov : bool, optional

Include model covariance when calculating chisq. Default is False.

nwalkers : int, optional

Number of walkers in the sampler.

nburn : int, optional

Number of samples in burn-in phase.

nsamples : int, optional

Number of samples in production run.

sampler: str, optional

The kind of sampler to use. Currently 'ensemble' for `emcee.EnsembleSampler` and 'pt' for `emcee.PTSampler` are supported.

ntemps : int, optional

If `sampler == 'pt'` the number of temperatures to use for the parallel tempered sampler.

thin : int, optional

Factor by which to thin samples in production run. Output samples array will have (nsamples/thin) samples.

a : float, optional

Proposal scale parameter passed to the sampler.

warn : bool, optional

Issue a warning when dropping bands outside the wavelength range of the model. Default is True.

New in version 1.5.0

Returns **res** : Result

Has the following attributes:

- `param_names`: All parameter names of model, including fixed.
- `parameters`: Model parameters, with varied parameters set to mean value in samples.
- `vparam_names`: Names of parameters varied. Order of parameters matches order of samples.
- `samples`: 2-d array with shape `(N, len(vparam_names))`. Order of parameters in each row matches order in `res.vparam_names`.
- `covariance`: 2-d array giving covariance, measured from samples. Order corresponds to `res.vparam_names`.
- `errors`: dictionary giving square root of diagonal of covariance matrix for varied parameters. Useful for `plot_lc`.
- `mean_acceptance_fraction`: mean acceptance fraction for all walkers in the sampler.
- `ndof`: Number of degrees of freedom (len(data) - len(vparam_names)). *New in version 1.5.0.*
- `data_mask`: Boolean array the same length as data specifying whether each observation was used. *New in version 1.5.0.*

est_model : *Model*

Copy of input model with varied parameters set to mean value in samples.

sncosmo.nest_lc

sncosmo.**nest_lc**(*data*, *model*, *vparam_names*, *bounds*, *guess_amplitude_bound=False*, *minsnr=5.0*, *priors=None*, *ppfs=None*, *npoints=100*, *method='single'*, *maxiter=None*, *maxcall=None*, *modelcov=False*, *rstate=None*, *verbose=False*, *warn=True*, ***kwargs*)

Run nested sampling algorithm to estimate model parameters and evidence.

Parameters **data** : `Table` or `ndarray` or `dict`

Table of photometric data. Must include certain columns. See the "Photometric Data" section of the documentation for required columns.

model : *Model*

The model to fit.

vparam_names : list

Model parameters to vary in the fit.

bounds : `dict`

Bounded range for each parameter. Bounds must be given for each parameter, with the exception of `t0`: by default, the minimum bound is such that the latest phase of the model lines up with the earliest data point and the maximum bound is such that the earliest phase of the model lines up with the latest data point.

guess_amplitude_bound : bool, optional

If true, bounds for the model's amplitude parameter are determined automatically based on the data and do not need to be included in `bounds`. The lower limit is set to zero and the upper limit is 10 times the amplitude "guess" (which is based on the highest-flux data point in any band). Default is False.

minsnr : float, optional

Minimum signal-to-noise ratio of data points to use when guessing amplitude bound. Default is 5.

priors : `dict`, optional

Prior probability distribution function for each parameter. The keys should be parameter names and the values should be callables that accept a float. If a parameter is not in the dictionary, the prior defaults to a flat distribution between the bounds.

ppfs : `dict`, optional

Prior percent point function (inverse of the cumulative distribution function) for each parameter. If a parameter is in this dictionary, the ppf takes precedence over a prior pdf specified in `priors`.

npoints : int, optional

Number of active samples to use. Increasing this value increases the accuracy (due to denser sampling) and also the time to solution.

method : {'classic', 'single', 'multi'}, optional

Method used to select new points. Choices are 'classic', single-ellipsoidal ('single'), multi-ellipsoidal ('multi'). Default is 'single'.

maxiter : int, optional

Maximum number of iterations. Iteration may stop earlier if termination condition is reached. Default is no limit.

maxcall : int, optional

Maximum number of likelihood evaluations. Iteration may stop earlier if termination condition is reached. Default is no limit.

modelcov : bool, optional

Include model covariance when calculating chisq. Default is False.

rstate : `RandomState`, optional

RandomState instance. If not given, the global random state of the `numpy.random` module will be used.

verbose : bool, optional

Print running evidence sum on a single line.

warn : bool, optional

Issue warning when dropping bands outside the model range. Default is True.

New in version 1.5.0

Returns **res** : Result

Attributes are:

- `niter`: total number of iterations
- `ncall`: total number of likelihood function calls
- `time`: time in seconds spent in iteration loop.
- `logz`: natural log of the Bayesian evidence Z.
- `logzerr`: estimate of uncertainty in logz (due to finite sampling)
- `h`: Bayesian information.
- `vparam_names`: list of parameter names varied.
- `samples`: 2-d `ndarray`, shape is (nsamples, nparameters). Each row is the parameter values for a single sample. For example, `samples[0, :]` is the parameter values for the first sample.
- `logprior`: 1-d `ndarray` (length=nsamples); log(prior volume) for each sample.
- `logl`: 1-d `ndarray` (length=nsamples); log(likelihood) for each sample.
- `weights`: 1-d `ndarray` (length=nsamples); Weight corresponding to each sample. The weight is proportional to the prior * likelihood for the sample.
- `parameters`: 1-d `ndarray` of weighted-mean parameter values from samples (including fixed parameters). Order corresponds to `model.param_names`.
- `covariance`: 2-d `ndarray` of parameter covariance; indicies correspond to order of `vparam_names`. Calculated from `samples` and `weights`.
- `errors`: OrderedDict of varied parameter uncertainties. Corresponds to square root of diagonal entries in covariance matrix.
- `ndof`: Number of degrees of freedom (len(data) - len(vparam_names)).
- `bounds`: Dictionary of bounds on varied parameters (including any automatically determined bounds).

- `data_mask`: Boolean array the same length as data specifying whether each observation was used. *New in version 1.5.0.*

estimated_model : *Model*

A copy of the model with parameters set to the values in `res.parameters`.

Convenience functions

select_data(data, index)	Convenience function for indexing photometric data with covariance.
chisq(data, model[, modelcov])	Calculate chisq statistic for the model, given the data.
flatten_result(res)	Turn a result from fit_lc into a simple dictionary of key, value pairs.

sncosmo.select_data

sncosmo.**select_data**(*data*, *index*)

Convenience function for indexing photometric data with covariance.

This is like `data[index]` on an astropy Table, but handles covariance columns correctly.

Parameters **data** : Table

Table of photometric data.

index : slice or array or int

Row selection to apply to table.

Returns Table

Examples

We have a small table of photometry with a covariance column and we want to select some rows based on a mask:

```
>>> data = Table([[1., 2., 3.],
...               ['a', 'b', 'c'],
...               [[1.1, 1.2, 1.3],
...                [2.1, 2.2, 2.3],
...                [3.1, 3.2, 3.3]]],
...               names=['time', 'x', 'cov'])
>>> mask = np.array([True, True, False])
```

Selecting directly on the table, the covariance column is not sliced in each row: it has shape (2, 3) when it should be (2, 2):

```
>>> data[mask]
<Table length=2>
  time   x    cov [3]
float64 str1  float64
------- ---- ----------
    1.0    a 1.1 .. 1.3
    2.0    b 2.1 .. 2.3
```

Using `select_data` solves this:

```
>>> sncosmo.select_data(data, mask)
<Table length=2>
  time   x    cov [2]
float64 str1  float64
------- ---- ----------
    1.0    a 1.1 .. 1.2
    2.0    b 2.1 .. 2.2
```

sncosmo.chisq

sncosmo.**chisq**(*data*, *model*, *modelcov=False*)

Calculate chisq statistic for the model, given the data.

Parameters **model** : *Model*

data : Table or ndarray or dict

Table of photometric data. Must include certain columns. See the "Photometric Data" section of the documentation for required columns.

modelcov : bool

Include model covariance? Calls `model.bandfluxcov` method instead of `model.bandflux`. The source in the model must therefore implement covariance.

Returns **chisq** : float

sncosmo.flatten_result

sncosmo.**flatten_result**(*res*)

Turn a result from fit_lc into a simple dictionary of key, value pairs.

Useful when saving results to a text file table, where structures like a covariance matrix cannot be easily written to a single table row.

Parameters **res** : Result

Result object from *fit_lc*.

Returns **flatres** : Result

Flattened result. Keys are all strings, values are one of: float, int, string), suitable for saving to a text file.

10.5.5 Plotting

Convenience functions for quick standard plots (requires matplotlib)

plot_lc([data, model, bands, zp, zpsys, ...])	Plot light curve data or model light curves.

sncosmo.plot_lc

sncosmo.**plot_lc**(*data=None*, *model=None*, *bands=None*, *zp=25.0*, *zpsys='ab'*, *pulls=True*, *xfigsize=None*, *yfigsize=None*, *figtext=None*, *model_label=None*, *errors=None*, *ncol=2*, *figtextsize=1.0*, *show_model_params=True*, *tighten_ylim=False*, *color=None*, *cmap=None*, *cmap_lims=(3000.0*, *10000.0)*, *fill_data_marker=None*, *fname=None*, *fill_percentiles=None*, ***kwargs*)

Plot light curve data or model light curves.

Parameters **data** : astropy `Table` or similar, optional

Table of photometric data. Must include certain column names. See the "Photometric Data" section of the documentation for required columns.

model : *Model* or list thereof, optional

If given, model light curve is plotted. If a string, the corresponding model is fetched from the registry. If a list or tuple of *Model*, multiple models are plotted.

model_label : str or list, optional

If given, model(s) will be labeled in a legend in the upper left subplot. Must be same length as model.

errors : dict, optional

Uncertainty on model parameters. If given, along with exactly one model, uncertainty will be displayed with model parameters at the top of the figure.

bands : list, optional

List of Bandpasses, or names thereof, to plot.

zp : float, optional

Zeropoint to normalize the flux in the plot (for the purpose of plotting all observations on a common flux scale). Default is 25.

zpsys : str, optional

Zeropoint system to normalize the flux in the plot (for the purpose of plotting all observations on a common flux scale). Default is `'ab'`.

pulls : bool, optional

If True (and if model and data are given), plot pulls. Pulls are the deviation of the data from the model divided by the data uncertainty. Default is `True`.

figtext : str, optional

Text to add to top of figure. If a list of strings, each item is placed in a separate "column". Use newline separators for multiple lines.

ncol : int, optional

Number of columns of axes. Default is 2.

xfigsize, yfigsize : float, optional

figure size in inches in x or y. Specify one or the other, not both. Default is to set axes panel size to 3.0 x 2.25 inches.

figtextsize : float, optional

Space to reserve at top of figure for figtext (if not None). Default is 1 inch.

show_model_params : bool, optional

If there is exactly one model plotted, the parameters of the model are added to `figtext` by default (as two additional columns) so that they are printed at the top of the figure. Set this to False to disable this behavior.

tighten_ylim : bool, optional

If true, tighten the y limits so that the model is visible (if any models are plotted).

color : str or mpl_color, optional

Color of data and model lines in each band. Can be any type of color that matplotlib understands. If None (default) a colormap will be used to choose a color for each band according to its central wavelength.

cmap : Colormap, optional

A matplotlib colormap to use, if color is None. If both color and cmap are None, a default colormap will be used.

cmap_lims : (float, float), optional

The wavelength limits for the colormap, in Angstroms. Default is (3000., 10000.), meaning that a bandpass with a central wavelength of 3000 Angstroms will be assigned a color at the low end of the colormap and a bandpass with a central wavelength of 10000 will be assigned a color at the high end of the colormap.

fill_data_marker : array_like, optional

Array of booleans indicating whether to plot a filled or unfilled marker for each data point. Default is all filled markers.

fname : str, optional

Filename to pass to savefig. If None (default), figure is returned.

fill_percentiles : (float, float, float), optional

When multiple models are given, the percentiles for a light curve confidence interval. The upper and lower perceniles define a fill between region, and the middle percentile defines a line that will be plotted over the fill between region.

kwargs : optional

Any additional keyword args are passed to `savefig`. Popular options include `dpi`, `format`, `transparent`. See matplotlib docs for full list.

Returns **fig** : matplotlib `Figure`

Only returned if `fname` is `None`. Display to screen with `plt.show()` or save with `fig.savefig(filename)`. When creating many figures, be sure to close with `plt.close(fig)`.

Examples

```
>>> import sncosmo
>>> import matplotlib.pyplot as plt
```

Load some example data:

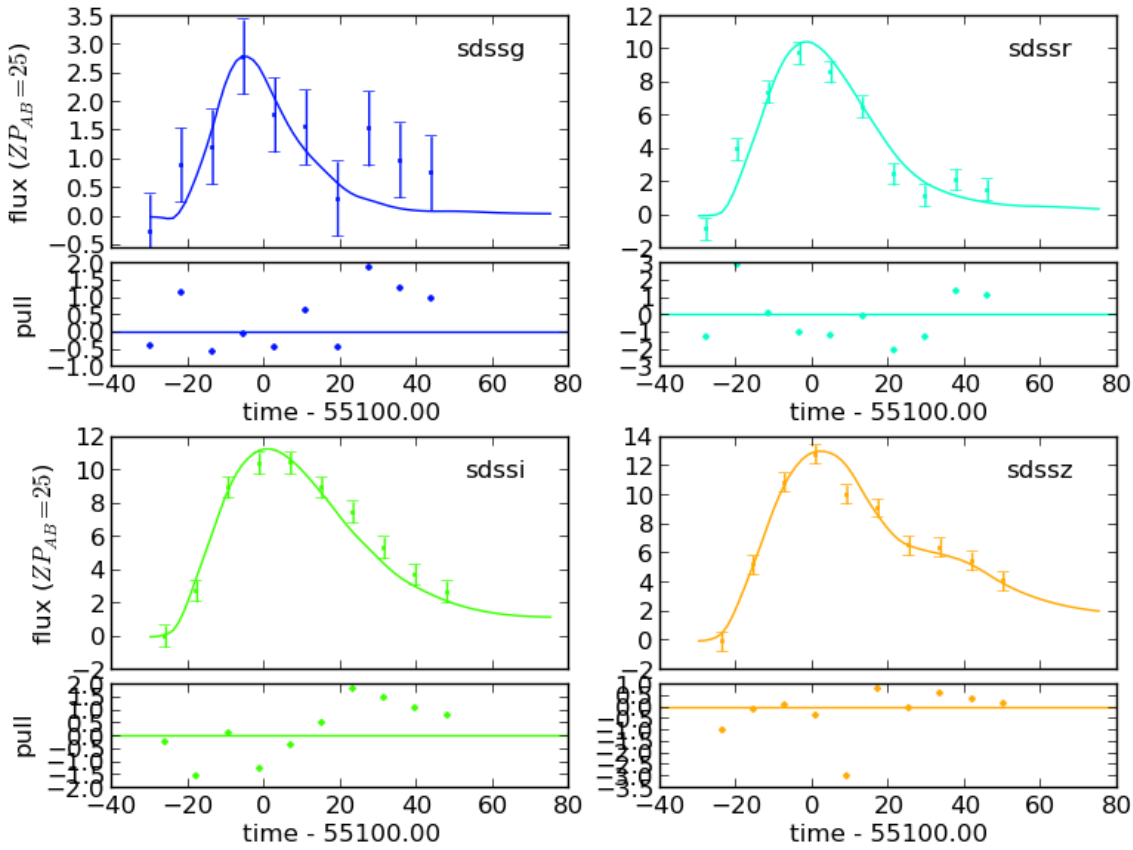
```
>>> data = sncosmo.load_example_data()
```

Plot the data, displaying to the screen:

```
>>> fig = plot_lc(data)
>>> plt.show()
```

Plot a model along with the data:

```
>>> model = sncosmo.Model('salt2')
>>> model.set(z=0.5, c=0.2, t0=55100., x0=1.547e-5)
>>> sncosmo.plot_lc(data, model=model)
```



Plot just the model, for selected bands:

```
>>> sncosmo.plot_lc(model=model,
...                 bands=['sdssg', 'sdssr'])
```

Plot figures on a multipage pdf:

```
>>> from matplotlib.backends.backend_pdf import PdfPages
>>> pp = PdfPages('output.pdf')
```

```
>>> # Do the following as many times as you like:
>>> sncosmo.plot_lc(data, fname=pp, format='pdf')
```

```
>>> # Don't forget to close at the end:
>>> pp.close()
```

10.5.6 Simulation

zdist(zmin, zmax[, time, area, ratefunc, ...])	Generate a distribution of redshifts.
realize_lcs(observations, model, params[, ...])	Realize data for a set of SNe given a set of observations.

sncosmo.zdist

sncosmo.**zdist**(*zmin*, *zmax*, *time=365.25*, *area=1.0*, *ratefunc=<function <lambda>>*, *cosmo=FlatLambdaCDM(H0=70 km / (Mpc s), Om0=0.3, Tcmb0=0 K, Neff=3.04, m_nu=None, Ob0=None)*)

Generate a distribution of redshifts.

Generates the correct redshift distribution and number of SNe, given the input volumetric SN rate, the cosmology, and the observed area and time.

Parameters **zmin, zmax** : float

Minimum and maximum redshift.

time : float, optional

Time in days (default is 1 year).

area : float, optional

Area in square degrees (default is 1 square degree). `time` and `area` are only used to determine the total number of SNe to generate.

ratefunc : callable

A callable that accepts a single float (redshift) and returns the comoving volumetric rate at each redshift in units of yr^-1 Mpc^-3. The default is a function that returns `1.e-4`.

cosmo : `Cosmology`, optional

Cosmology used to determine volume. The default is a FlatLambdaCDM cosmology with `Om0=0.3`, `H0=70.0`.

Examples

Loop over the generator:

```
>>> for z in zdist(0.0, 0.25):
...     print(z)
...
0.151285827576
0.204078030595
0.201009196731
0.181635472172
0.17896188781
0.226561237264
0.192747368762
```

This tells us that in one observer-frame year, over 1 square degree, 7 SNe occured at redshifts below 0.35 (given the default volumetric SN rate of 10^-4 SNe yr^-1 Mpc^-3). The exact number is drawn from a Poisson distribution.

Generate the full list of redshifts immediately:

```
>>> zlist = list(zdist(0., 0.25))
```

Define a custom volumetric rate:

```
>>> def snrate(z):
...     return 0.5e-4 * (1. + z)
...
>>> zlist = list(zdist(0., 0.25, ratefunc=snrate))
```

sncosmo.realize_lcs

sncosmo.**realize_lcs**(*observations*, *model*, *params*, *thresh=None*, *trim_observations=False*, *scatter=True*)

Realize data for a set of SNe given a set of observations.

Parameters **observations** : Table or ndarray

Table of observations. Must contain the following column names: band, time, zp, zpsys, gain, skynoise.

model : *sncosmo.Model*

The model to use in the simulation.

params : list (or generator) of dict

List of parameters to feed to the model for realizing each light curve.

thresh : float, optional

If given, light curves are skipped (not returned) if none of the data points have signal-to-noise greater than thresh.

trim_observations : bool, optional

If True, only observations with times between model.mintime() and model.maxtime() are included in result table for each SN. Default is False.

scatter : bool, optional

If True, the flux value of the realized data is calculated by adding a random number drawn from a Normal Distribution with a standard deviation equal to the fluxerror of the observation to the bandflux value of the observation calculated from model. Default is True.

Returns **sne** : list of Table

Table of realized data for each item in params.

Notes

skynoise is the image background contribution to the flux measurement error (in units corresponding to the specified zeropoint and zeropoint system). To get the error on a given measurement, skynoise is added in quadrature to the photon noise from the source.

It is left up to the user to calculate skynoise as they see fit as the details depend on how photometry is done and possibly how the PSF is is modeled. As a simple example, assuming a Gaussian PSF, and perfect PSF photometry, skynoise would be 4 * pi * sigma_PSF * sigma_pixel where sigma_PSF is the standard deviation of the PSF in pixels and sigma_pixel is the background noise in a single pixel in counts.

10.5.7 Registry

Register and retrieve custom built-in sources, bandpasses, and magnitude systems

register(instance[, name, data_class, force])	Register a class instance.
register_loader(data_class, name, func[, ...])	Register a data reading function.
get_source(name[, version, copy])	Retrieve a Source from the registry by name.
get_bandpass(name, *args)	Get a Bandpass from the registry by name.
get_magsystem(name)	Get a MagSystem from the registry by name.

sncosmo.register

sncosmo.**register**(*instance*, *name=None*, *data_class=None*, *force=False*)

Register a class instance.

Note: Formerly accessed as `sncosmo.registry.register` prior to v1.2.

Parameters **instance** : object

The object to be registered.

name : str, optional

Identifier. If `None`, the name is taken from the `name` attribute of the instance, if it exists and is a string.

data_class : classobj, optional

If given, the instance is registered as an instance of this class rather the the class of the instance itself. Use this for registering subclasses when you wish them to be accessible from their superclass.

force : bool, optional

Whether to override any existing instance of the same name. Note: this may not play well with versioned instances.

sncosmo.register_loader

sncosmo.**register_loader**(*data_class*, *name*, *func*, *args=None*, *version=None*, *meta=None*, *force=False*)

Register a data reading function.

Note: Formerly accessed as `sncosmo.registry.register_loader` prior to v1.2.

Parameters **data_class** : classobj

The class of the object that the loader returns.

name : str

The data identifier.

func : callable

The function to read in the data. Must accept a name and version keyword argument.

args : list, optional

Arguments to pass to the function. Default is an empty list.

version : str, optional

Sub-version of name, if desired. Use formats such as `'1'`, `'1.0'`, `'1.0.0'`, etc. Default is `None`.

force : bool, optional

Whether to override any existing function if already present.

meta : dict, optional

Metadata describing this loader. Default is an empty dictionary.

sncosmo.get_source

`sncosmo.`**`get_source`**(*name*, *version=None*, *copy=False*)

Retrieve a Source from the registry by name.

Parameters **name** : str

Name of source in the registry.

version : str, optional

Version identifier for sources with multiple versions. Default is `None` which corresponds to the latest, or only, version.

copy : bool, optional

If True and if `name` is already a Source instance, return a copy of it. (If `name` is a str a copy of the instance in the registry is always returned, regardless of the value of this parameter.) Default is False.

sncosmo.get_bandpass

`sncosmo.`**`get_bandpass`**(*name*, **args*)

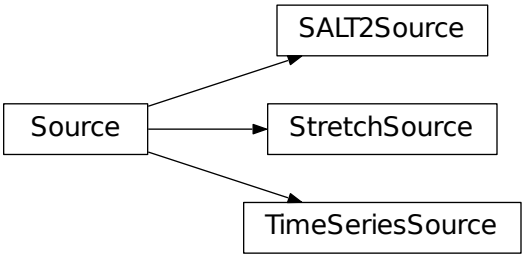
Get a Bandpass from the registry by name.

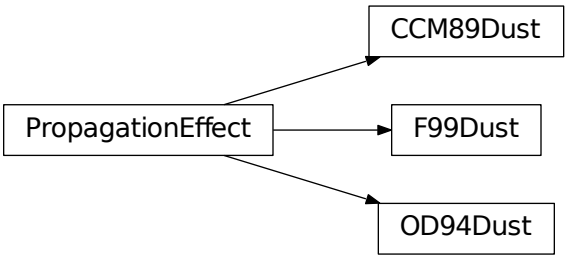
sncosmo.get_magsystem

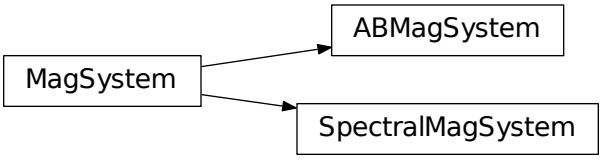
`sncosmo.`**`get_magsystem`**(*name*)

Get a MagSystem from the registry by name.

10.5.8 Class Inheritance Diagrams







10.6 List of Built-in Sources

Name	Version	Type	Subclass	Reference	Website	Notes
'nugent-sn1a'	'1.2'	SN Ia	*TimeSeriesSource*	*[N023]*	a	
'nugent-sn91t'	'1.1'	SN Ia	*TimeSeriesSource*	*[S043]*	a	

Continued on next page

Table 10.42 – continued from previous page

Name	Version	Type	Subclass	Reference	Website	Notes
'nugent-sn91bg'	'1.1'	SN Ia	TimeSeriesSource	[N023]	a	
'nugent-sn1bc'	'1.1'	SN Ib/c	TimeSeriesSource	[L053]	a	
'nugent-hyper'	'1.2'	SN Ib/c	TimeSeriesSource	[L053]	a	
'nugent-sn2p'	'1.2'	SN IIP	TimeSeriesSource	[G993]	a	
'nugent-sn2l'	'1.2'	SN IIL	TimeSeriesSource	[G993]	a	
'nugent-sn2n'	'2.1'	SN IIn	TimeSeriesSource	[G993]	a	
's11-2004hx'	'1.0'	SN IIL/P	TimeSeriesSource	[S113]	b	[R4]
's11-2005lc'	'1.0'	SN IIP	TimeSeriesSource	[S113]	b	[R4]
's11-2005hl'	'1.0'	SN Ib	TimeSeriesSource	[S113]	b	[R4]
's11-2005hm'	'1.0'	SN Ib	TimeSeriesSource	[S113]	b	[R4]
's11-2005gi'	'1.0'	SN IIP	TimeSeriesSource	[S113]	b	[R4]
's11-2006fo'	'1.0'	SN Ic	TimeSeriesSource	[S113]	b	[R4]
's11-2006jo'	'1.0'	SN Ib	TimeSeriesSource	[S113]	b	[R4]
's11-2006jl'	'1.0'	SN IIP	TimeSeriesSource	[S113]	b	[R4]
'hsiao'	'1.0'	SN Ia	TimeSeriesSource	[H073]	c	[R5]
'hsiao'	'2.0'	SN Ia	TimeSeriesSource	[H073]	c	[R5]
'hsiao'	'3.0'	SN Ia	TimeSeriesSource	[H073]	c	[R5]
'hsiao-subsampled'	'3.0'	SN Ia	TimeSeriesSource	[H073]	c	[R5]
'salt2'	'2.0'	SN Ia	SALT2Source	[G103]	d	
'salt2'	'2.4'	SN Ia	SALT2Source	[B14b3]	d	
'salt2-extended'	'1.0'	SN Ia	SALT2Source		b	[R6]
'snf-2011fe'	'1.0'	SN Ia	TimeSeriesSource	[P133]	e	
'snana-2004fe'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-2004gq'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-sdss004012'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-2006fo'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-sdss014475'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-2006lc'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-2007ms'	'1.0'	SN II-pec	TimeSeriesSource		f	[R7]
'snana-04d1la'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-04d4jv'	'1.0'	SN Ic	TimeSeriesSource		f	[R7]
'snana-2004gv'	'1.0'	SN Ib	TimeSeriesSource		f	[R7]
'snana-2006ep'	'1.0'	SN Ib	TimeSeriesSource		f	[R7]
'snana-2007y'	'1.0'	SN Ib	TimeSeriesSource		f	[R7]
'snana-2004ib'	'1.0'	SN Ib	TimeSeriesSource		f	[R7]
'snana-2005hm'	'1.0'	SN Ib	TimeSeriesSource		f	[R7]
'snana-2006jo'	'1.0'	SN Ib	TimeSeriesSource		f	[R7]
'snana-2007nc'	'1.0'	SN Ib	TimeSeriesSource		f	[R7]
'snana-2004hx'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2005gi'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2006gq'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2006kn'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2006jl'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2006iw'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2006kv'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2006ns'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2007iz'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2007nr'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]
'snana-2007kw'	'1.0'	SN IIP	TimeSeriesSource		f	[R7]

Continued on next page

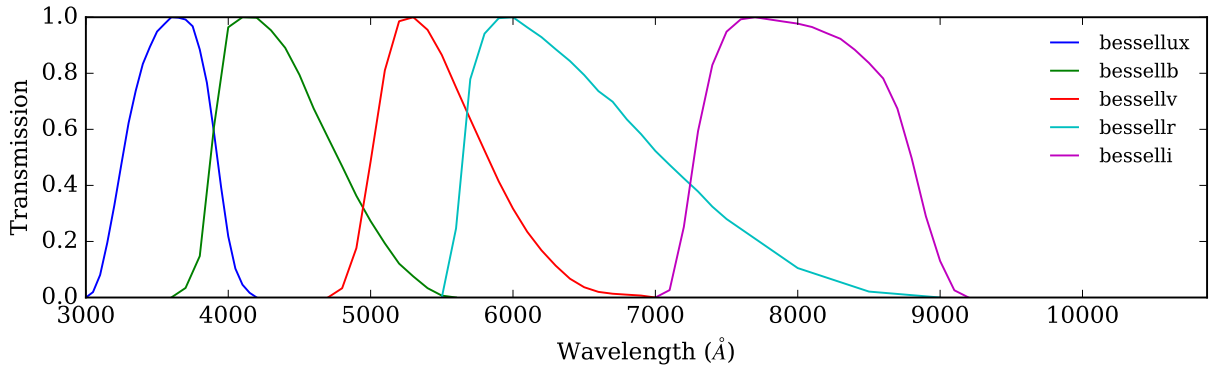
Table 10.42 – continued from previous page

Name	Version	Type	Subclass	Reference	Website	Notes
'snana-2007ky'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007lj'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007lb'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007ll'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007nw'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007ld'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007md'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007lz'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007lx'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007og'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007ny'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007nv'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2007pg'	'1.0'	SN IIP	*TimeSeriesSource*		f	*[R7]*
'snana-2006ez'	'1.0'	SN IIn	*TimeSeriesSource*		f	*[R7]*
'snana-2006ix'	'1.0'	SN IIn	*TimeSeriesSource*		f	*[R7]*
'whalen-z15b'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'whalen-z15d'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'whalen-z15g'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'whalen-z25b'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'whalen-z25d'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'whalen-z25g'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'whalen-z40b'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'whalen-z40g'	'1.0'	PopIII	*TimeSeriesSource*	*[Whalen133]*		*[R8]*
'mlcs2k2'	'1.0'	SN Ia	`MLCS2k2Source`	*[Jha073]*		*[R9]*

10.7 List of Built-in Bandpasses

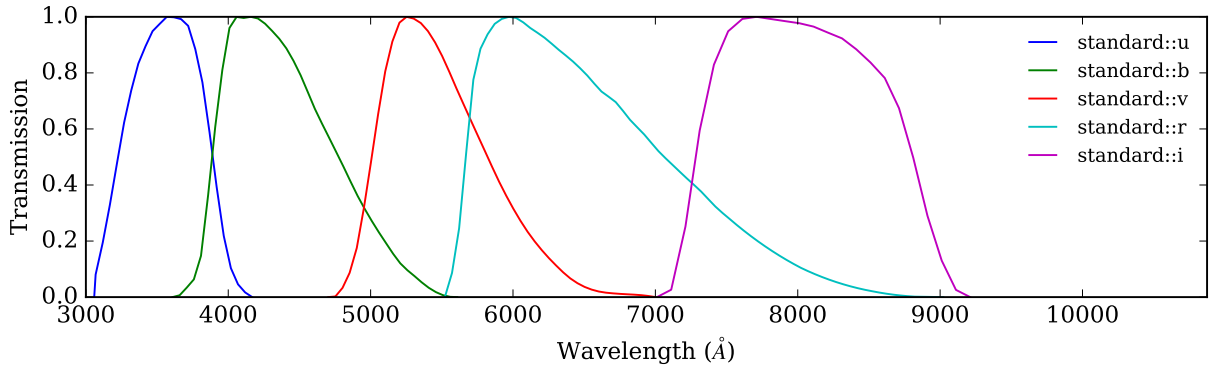
10.7.1 bessell

Name	Description	Reference	Data URL	Retrieved
'bessellux'	Representation of Johnson-Cousins UBVRI system	*[B900]*		
'besselllb'	Representation of Johnson-Cousins UBVRI system	*[B900]*		
'bessellv'	Representation of Johnson-Cousins UBVRI system	*[B900]*		
'bessellr'	Representation of Johnson-Cousins UBVRI system	*[B900]*		
'besselli'	Representation of Johnson-Cousins UBVRI system	*[B900]*		



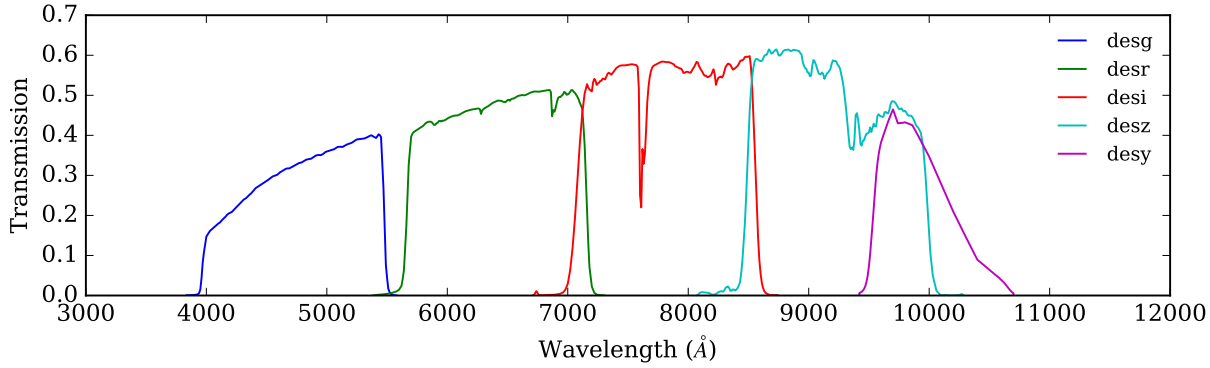
10.7.2 snls3-landolt

Name	Description	Reference	Data URL	Retrieved
'standard::u'	Bessell bandpasses shifted as in JLA analysis	*[B14a0]*	a	13 February 2017
'standard::b'	Bessell bandpasses shifted as in JLA analysis	*[B14a0]*	a	13 February 2017
'standard::v'	Bessell bandpasses shifted as in JLA analysis	*[B14a0]*	a	13 February 2017
'standard::r'	Bessell bandpasses shifted as in JLA analysis	*[B14a0]*	a	13 February 2017
'standard::i'	Bessell bandpasses shifted as in JLA analysis	*[B14a0]*	a	13 February 2017



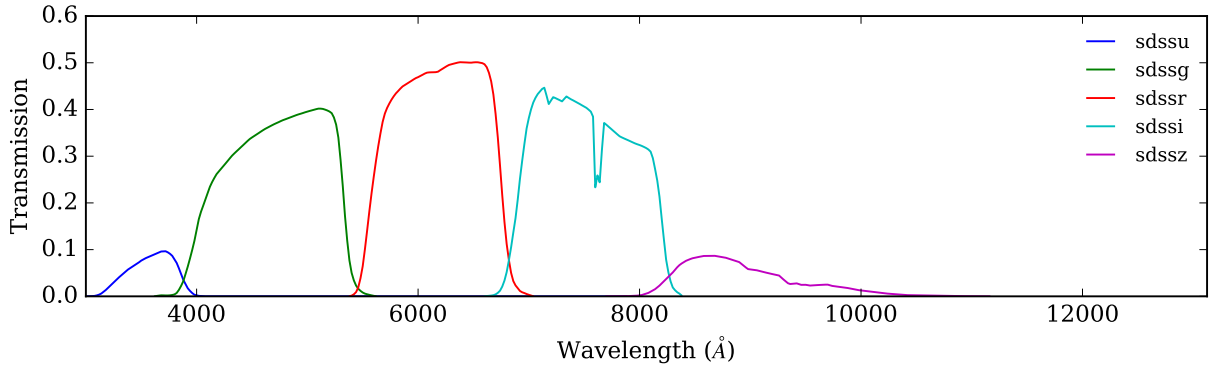
10.7.3 des

Name	Description	Reference	Data URL	Retrieved
'desg'	Dark Energy Camera grizy filter set at airmass 1.3			22 March 2013
'desr'	Dark Energy Camera grizy filter set at airmass 1.3			22 March 2013
'desi'	Dark Energy Camera grizy filter set at airmass 1.3			22 March 2013
'desz'	Dark Energy Camera grizy filter set at airmass 1.3			22 March 2013
'desy'	Dark Energy Camera grizy filter set at airmass 1.3			22 March 2013



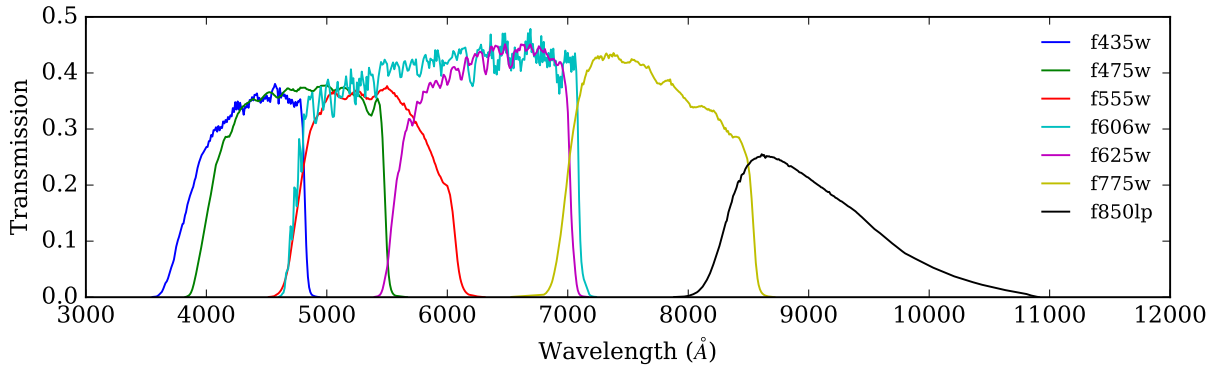
10.7.4 sdss

Name	Description	Reference	Data URL	Retrieved
'sdssu'	SDSS 2.5m imager at airmass 1.3 (including atmosphere), normalized	*[D100]*		
'sdssg'	SDSS 2.5m imager at airmass 1.3 (including atmosphere), normalized	*[D100]*		
'sdssr'	SDSS 2.5m imager at airmass 1.3 (including atmosphere), normalized	*[D100]*		
'sdssi'	SDSS 2.5m imager at airmass 1.3 (including atmosphere), normalized	*[D100]*		
'sdssz'	SDSS 2.5m imager at airmass 1.3 (including atmosphere), normalized	*[D100]*		



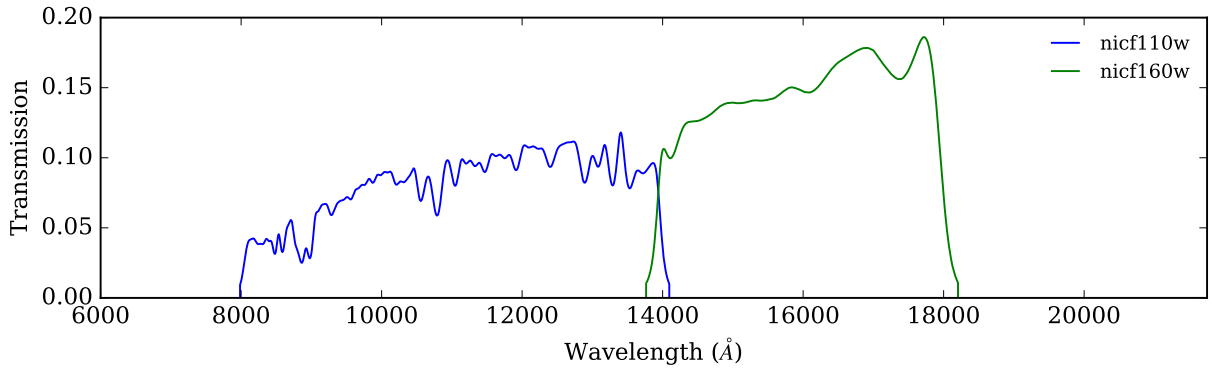
10.7.5 acs

Name	Description	Reference	Data URL	Retrieved
'f435w'	Hubble Space Telescope ACS WFC filters		b	direct download
'f475w'	Hubble Space Telescope ACS WFC filters		b	direct download
'f555w'	Hubble Space Telescope ACS WFC filters		b	direct download
'f606w'	Hubble Space Telescope ACS WFC filters		b	direct download
'f625w'	Hubble Space Telescope ACS WFC filters		b	direct download
'f775w'	Hubble Space Telescope ACS WFC filters		b	direct download
'f850lp'	Hubble Space Telescope ACS WFC filters		b	direct download



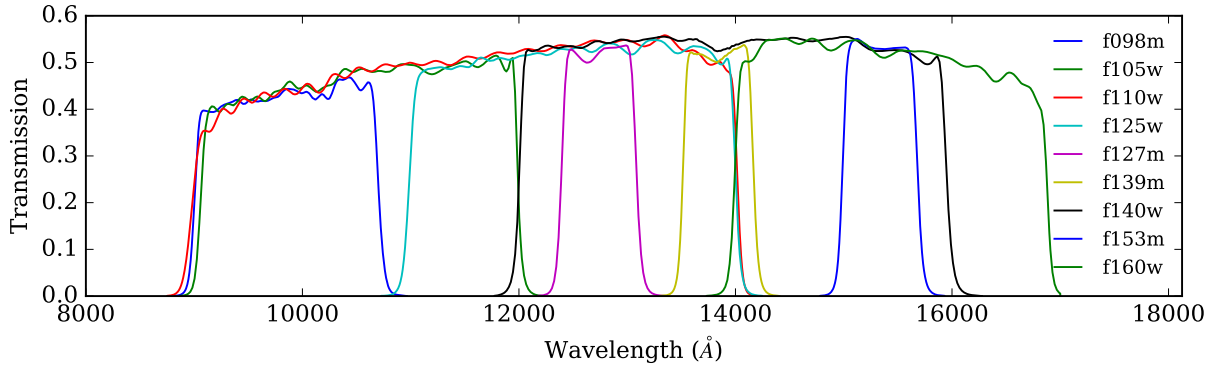
10.7.6 nicmos-nic2

Name	Description	Reference	Data URL	Retrieved
'nicf110w'	Hubble Space Telescope NICMOS2 filters		c	05 Aug 2014
'nicf160w'	Hubble Space Telescope NICMOS2 filters		c	05 Aug 2014



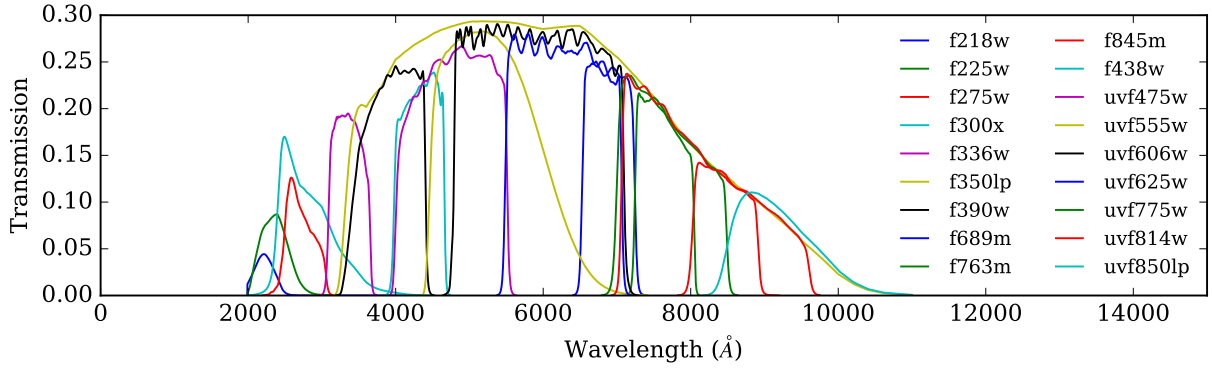
10.7.7 wfc3-ir

Name	Description	Reference	Data URL	Retrieved
'f098m'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f105w'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f110w'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f125w'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f127m'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f139m'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f140w'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f153m'	Hubble Space Telescope WFC3 IR filters		d	direct download
'f160w'	Hubble Space Telescope WFC3 IR filters		d	direct download



10.7.8 wfc3-uvis

Name	Description	Reference	Data URL	Retrieved
'f218w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f225w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f275w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f300x'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f336w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f350lp'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f390w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f689m'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f763m'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f845m'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'f438w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'uvf475w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'uvf555w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'uvf606w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'uvf625w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'uvf775w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'uvf814w'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download
'uvf850lp'	Hubble Space Telescope WFC3 UVIS filters (CCD 2)		d	direct download

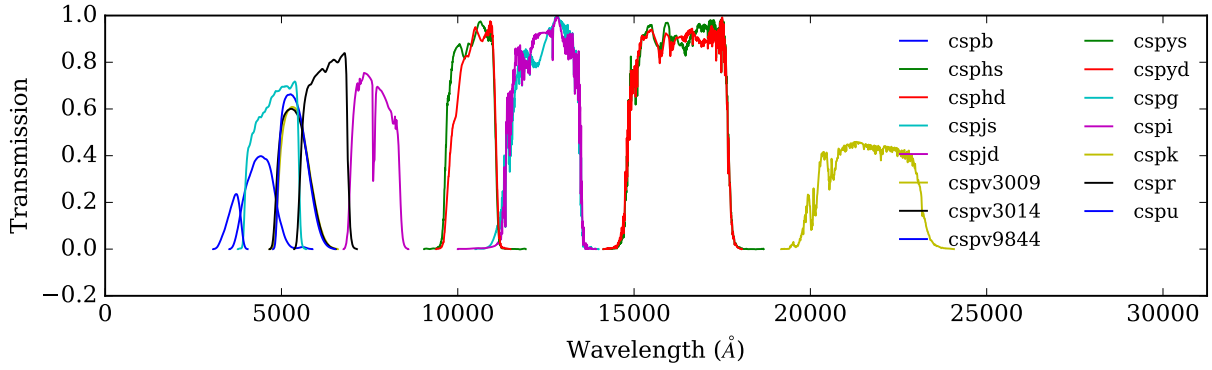


10.7.9 kepler

Name	Description	Reference	Data URL	Retrieved
'kepler'	Bandpass for the Kepler spacecraft		e	direct download

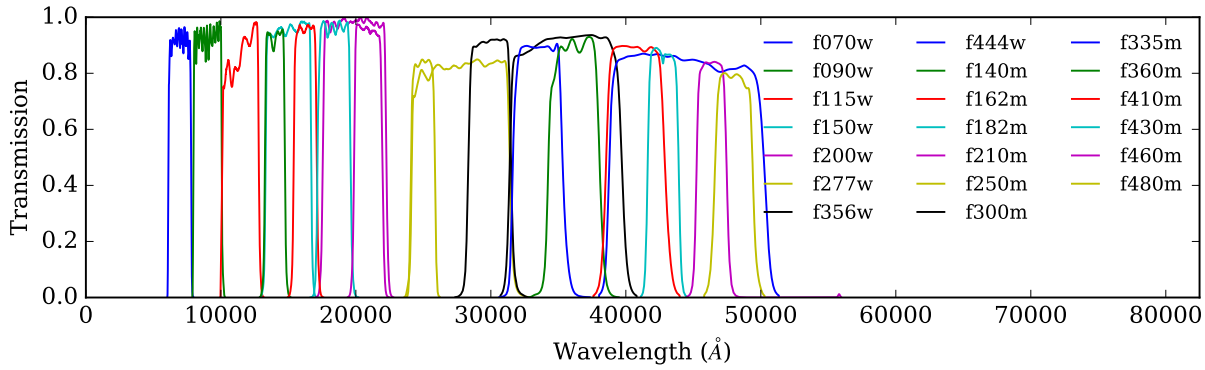
10.7.10 csp

Name	Description	Reference	Data URL	Retrieved
'cspb'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'csphs'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'csphd'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspjs'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspjd'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspv3009'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspv3014'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspv9844'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspys'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspyd'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspg'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspi'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspk'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspr'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017
'cspu'	Carnegie Supernova Project filters (Swope+DuPont Telescopes) updated 6 Oct 2016		f	8 Feb 2017



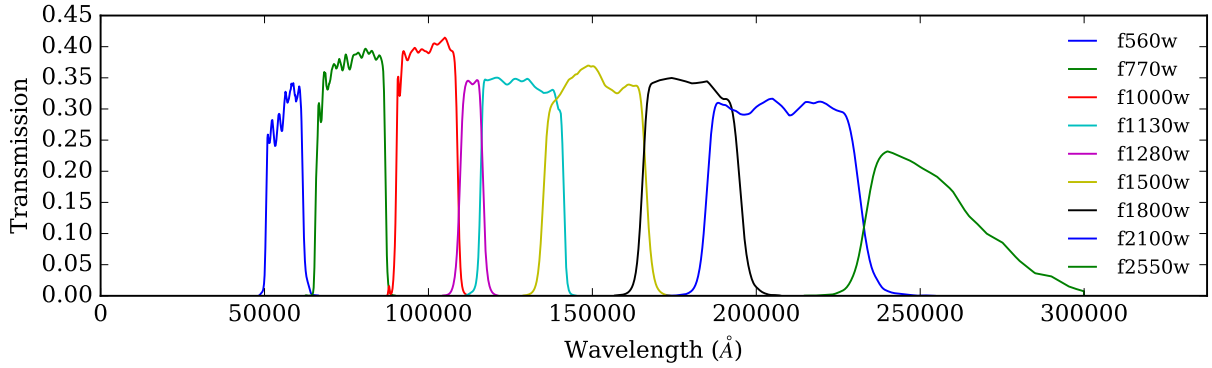
10.7.11 jwst-nircam

Name	Description	Reference	Data URL	Retrieved
'f070w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f090w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f115w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f150w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f200w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f277w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f356w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f444w'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f140m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f162m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f182m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f210m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f250m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f300m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f335m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f360m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f410m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f430m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f460m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014
'f480m'	James Webb Space Telescope NIRCAM Wide+Medium filters		g	09 Sep 2014



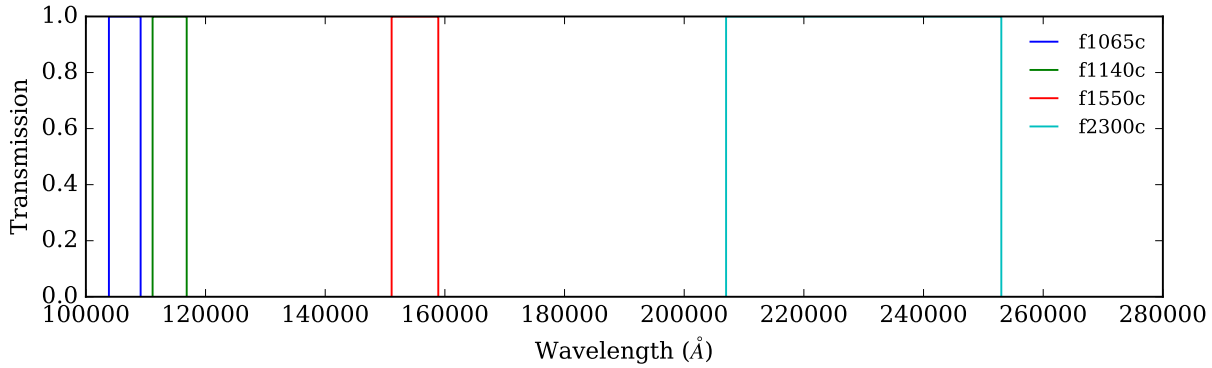
10.7.12 jwst-miri

Name	Description	Reference	Data URL	Retrieved
'f560w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f770w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f1000w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f1130w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f1280w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f1500w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f1800w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f2100w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017
'f2550w'	James Webb Space Telescope MIRI filters		h	16 Feb 2017



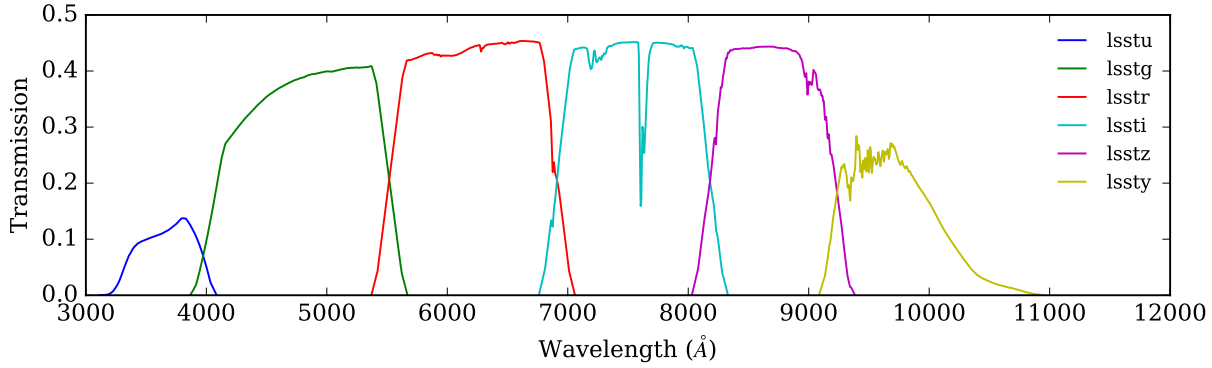
10.7.13 jwst-miri-tophat

Name	Description	Refer-ence	Data URL	Retrieved
'f1065c'	James Webb Space Telescope MIRI filters (idealized tophat)		i	09 Sep 2014
'f1140c'	James Webb Space Telescope MIRI filters (idealized tophat)		i	09 Sep 2014
'f1550c'	James Webb Space Telescope MIRI filters (idealized tophat)		i	09 Sep 2014
'f2300c'	James Webb Space Telescope MIRI filters (idealized tophat)		i	09 Sep 2014



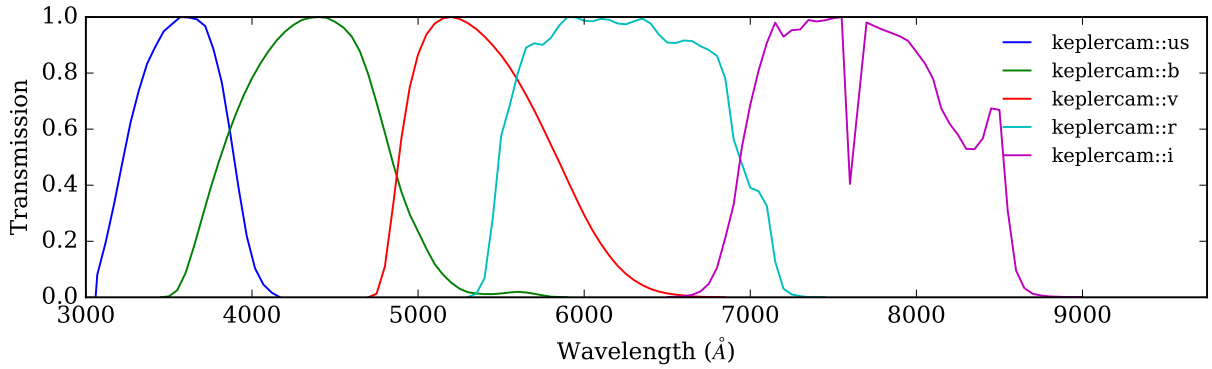
10.7.14 lsst

Name	Description	Reference	Data URL	Retrieved
'lsstu'	LSST baseline total throughputs, v1.1.		j	16 Nov 2016
'lsstg'	LSST baseline total throughputs, v1.1.		j	16 Nov 2016
'lsstr'	LSST baseline total throughputs, v1.1.		j	16 Nov 2016
'lssti'	LSST baseline total throughputs, v1.1.		j	16 Nov 2016
'lsstz'	LSST baseline total throughputs, v1.1.		j	16 Nov 2016
'lssty'	LSST baseline total throughputs, v1.1.		j	16 Nov 2016



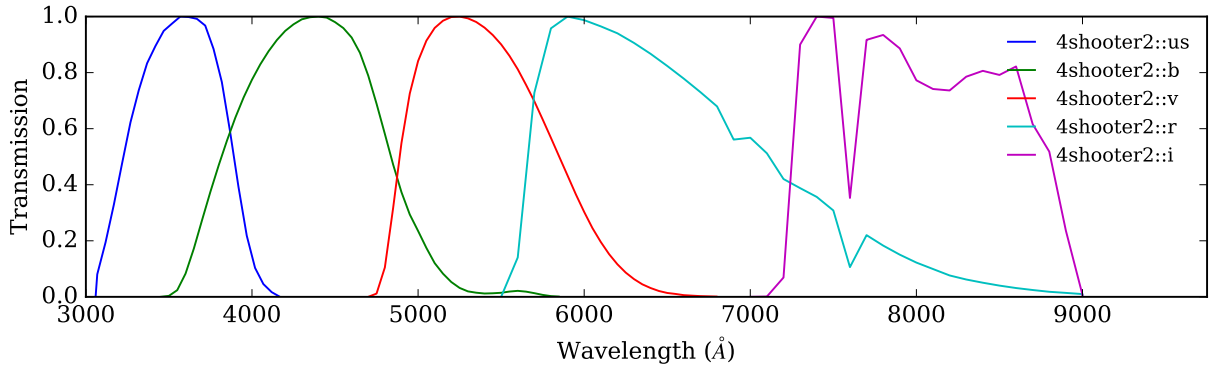
10.7.15 keplercam

Name	Description	Reference	Data URL	Retrieved
'keplercam::us'	Keplercam transmissions as used in JLA		a	13 Feb 2017
'keplercam::b'	Keplercam transmissions as used in JLA		a	13 Feb 2017
'keplercam::v'	Keplercam transmissions as used in JLA		a	13 Feb 2017
'keplercam::r'	Keplercam transmissions as used in JLA		a	13 Feb 2017
'keplercam::i'	Keplercam transmissions as used in JLA		a	13 Feb 2017



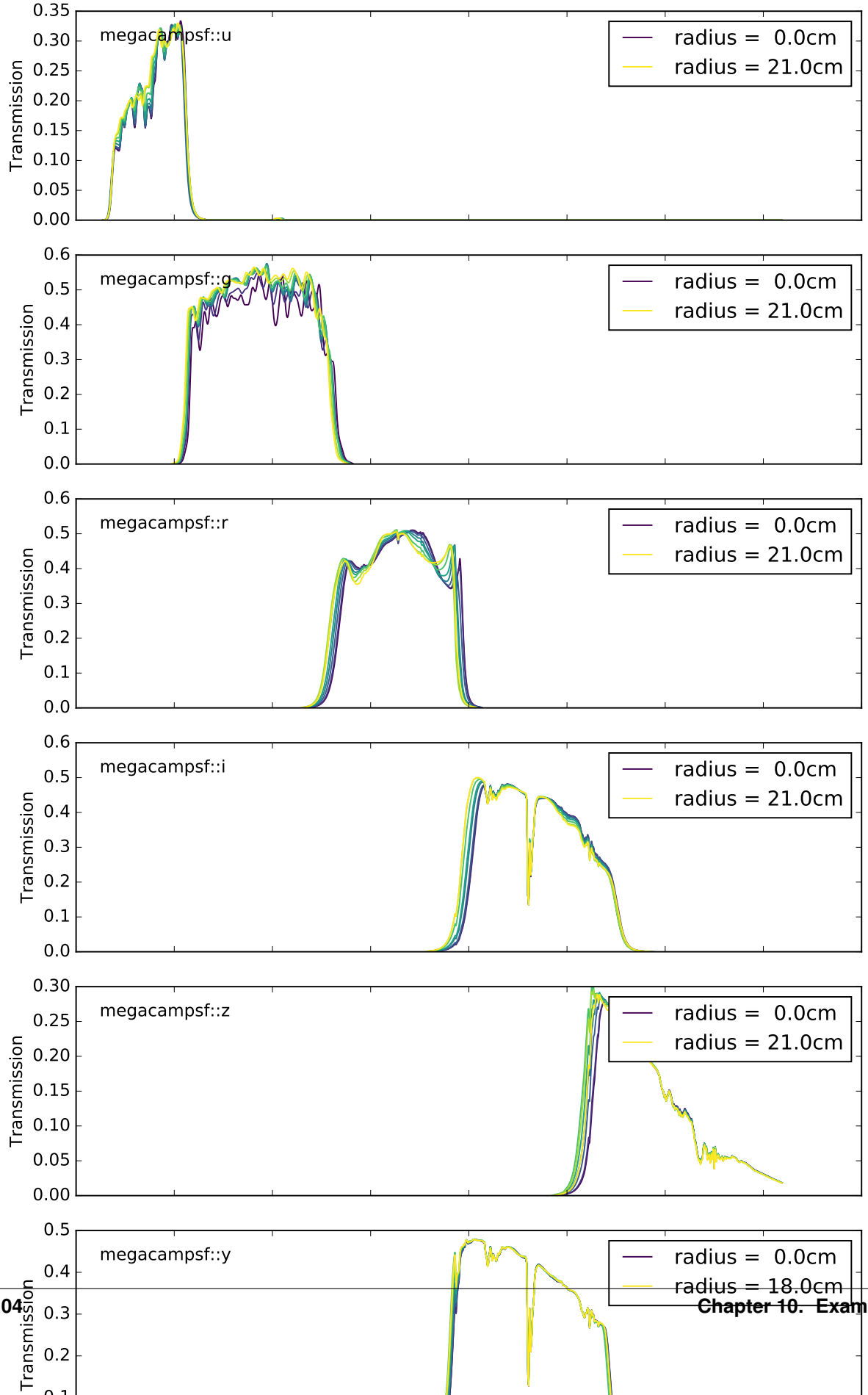
10.7.16 4shooter2

Name	Description	Reference	Data URL	Retrieved
'4shooter2::us'	4Shooter filters as used in JLA		a	13 Feb 2017
'4shooter2::b'	4Shooter filters as used in JLA		a	13 Feb 2017
'4shooter2::v'	4Shooter filters as used in JLA		a	13 Feb 2017
'4shooter2::r'	4Shooter filters as used in JLA		a	13 Feb 2017
'4shooter2::i'	4Shooter filters as used in JLA		a	13 Feb 2017



10.7.17 megacampsf

These are radially-variable bandpasses. To get a Bandpass at a given radius, use `band = sncosmo.get_bandpass('megacampsf::g', 13.0)`


megacampsf::u
megacampsf::g
megacampsf::r
megacampsf::i
megacampsf::z
megacampsf::y
Transmission
radius = 0.0cm
radius = 21.0cm
radius = 18.0cm

10.8 List of Built-in Magnitude Systems

Name	Description	Subclass	Spectrum Source
'jla1'	JLA1 magnitude system based on BD+17 STIS v003 spectrum	*CompositeMagSystem*	a
'ab'	Source of 3631 Jy has magnitude 0 in all bands	*ABMagSystem*	
'vega'	Vega (alpha lyrae) has magnitude 0 in all bands.	*SpectralMagSystem*	b
'bd17'	BD+17d4708 has magnitude 0 in all bands.	*SpectralMagSystem*	b
'csp'	Carnegie Supernova Project magnitude system.	*CompositeMagSystem*	c
'ab-b12'	Betoule et al (2012) calibration of SDSS system.	*CompositeMagSystem*	a

CHAPTER 11

Reference / API

CHAPTER 12

More...

12.1 Version History

Note: SNCosmo uses Semantic Versioning for its version numbers. Specifically, this means that code written for sncosmo v1.0 will continue to work with any v1.x version. However, exact results may differ between versions in the 1.x series. (For example, due to changes in integration method.)

12.1.1 v1.5.3 (2017-11-24)

- Fix pickling issue with Cython v0.26.

12.1.2 v1.5.2 (2017-11-24)

- Improve numeric stability by using pseudo-inverse instead of inverse in likelihood calculation.
- Remote data download fixes.

12.1.3 v1.5.1 (2017-05-12)

No new features or bugfixes. This release simply removes build dependencies to enable building conda packages on conda-forge.

12.1.4 v1.5.0 (2017-04-20)

This is a major new release. The highlight is really close compatibility of the SALT2 model and fitting procedure with `snfit`, the "official" SALT2 fitter.

- `SALT2Source`: Internal interpolation scheme of `SALT2Source` updated to match `snfit` implementation exactly. Test suite now tests against `snfit` implementation.

- `fit_lc()`:
  - Handling of model covariance updated to match that of `snfit`: model covariance is fixed for each fit and fit is repeated until convergence.
  - New arguments `phase_range` and `wave_range`. If given, data outside this range will be discarded after an initial fit and additional fits will be performed until convergence. With `phase_range=(-15., 45.)` and `wave_range=(3000., 7000.)`, behavior approximates that of snfit with default arguments.
  - Added support for covariance in photometric data measurements, and this covariance is used in `fit_lc()` if present. Covariance is stored as a `'fluxcov'` column in the table of measurements.
  - Result includes two new attributes: `data_mask`, a boolean array indicating which rows in the input data were used in the final fit (since multiple fits might be performed), and `nfit`, the number of fits performed.
  - New argument `warn` can be set to False to turn off warnings about dropping bands outside model wavelength range.
- `read_lc()`:
  - Added support for reading snfit-format "covmat" files into a table of photometry:

```
>>> data = read_lc('filename', format='salt2', read_covmat=True)
>>> data['Fluxcov'].shape == (len(data), len(data))
True
```

  - New keyword argument `expand_bands`. When True, the returned band column will contain `Bandpass` objects instead of strings. (Strings converted to bandpass objects using `sncosmo.get_bandpass()`.) This is particularly useful for position-dependent bandpasses in the salt2 file format, such as `megacampsf`: `read_lc()` reads the position from the header and feeds the position to `get_bandpass()` to get a Bandpass object for the correct position.
- Built-in bandpasses and magnitude systems: Many new built-in bandpasses and magnitude systems.
- Configuration: The environment variable `SNCOSMO_DATA_DIR` can be used to set the path to the data directory. If set, it takes precedence over the `data_dir` variable in the configuration file (`$HOME/.astropy/config/sncosmo.cfg`).

12.1.5 v1.4.0 (2016-11-16)

- `SFD98Map` and `get_ebv_from_map` deprecated in favor of separate package sfdmap which has vastly improved performance (200x faster) for the typical case of scalar coordinates in ICRS frame.
- `animate_source()` deprecated. This is a "fun extra" that is difficult to test and no longer seems to work.
- Cython implementation of extinction functions has been factored out into a separate Python module called `extinction`, which is now a dependency.
- `Model.bandflux()` and `Source.bandflux()` now integrate on a fixed wavelength grid of 5 angstroms regardless of the wavelength grid of the bandpass. This will result in small differences in results from previous sncosmo versions.
- The internal (publicly undocumented) `Spectrum` class now acts more like `Model`; in particular, its `bandflux()` method now behaves the same way. As `Spectrum` backs `SpectralMagSystem`, this makes the integration of models and zeropoint spectra more consistent.
- Experimental (non-public) support for aliases for bandpasses, such as `'SDSS::g'` for `'sdssg'`.
- Sources now use cubic rather than quadratic spline interpolation internally.

- `Model.source_peakmag()` and `Model.set_source_peakmag()` added as convenience functions for `Model.source.peakmag()` and `Model.source.set_peakmag()` respectively.
- **[Bugfix]** Fixed missing import of `math` module in `mcmc_lc()` when using the `priors` keyword. [Backported to v1.3.1] [#143]

12.1.6 v1.3.0 (2016-06-30)

This is mostly a bugfix release, but it also **drops support for Python 2.6.** Python 2.7 is now the minimum supported Python version.

- Updates for compatibility with AstroPy 1.2.
- The registry now handles subclasses more robustly. For example, if `magsys` is an instance of `SpectralMagSystem`, the following used to fail:

```
sncosmo.register(magsys, 'name')
sncosmo.get_magsystem('name')
```

  Now this works. [#132]
- **[Bugfix]** `SALT2Source` had a bug under Python 3 (only) yielding drastically wrong fluxes. Python 2 was not affected. [#138]

12.1.7 v1.2.0 (2015-12-01)

- **[API change]** Registry functions moved to the top-level namespace, as follows:
  - `sncosmo.registry.register()` -> `sncosmo.register()`
  - `sncosmo.registry.register_loader()` -> `sncosmo.register_loader()`
  - `sncosmo.registry.retrieve()` -> deprecated, use class-specific functions such as `sncosmo.get_bandpass()`.

  The old import paths will still work for backwards compatibility.
- `nest_lc()` now uses the `nestle` module under the hood. A new keyword `method` is available which selects different sampling methods implemented by `nestle`. The new methods provide potential efficiency gains.
- The MLCS2k2 model is now available as a built-in Source, with the name `'mlcs2k2'`.
- Bandpasses from the Carnegie Supernova Project added to built-ins.
- In `realize_lcs()`, a new `scatter` keyword makes adding noise optional.
- **[Bugfix]** Fix built-in Bessell bandpass definitions, which were wrong by a term proportional to inverse wavelength. This was due to misinterpretation of the trasmission units. [backported to v1.1.1] [#111]

12.1.8 v1.1.0 (2015-08-12)

This is a mostly bugfix release with more solid support for Python 3.

- Added `Model.color()` method.
- Remove `loglmax` from result of `nest_lc()`, which was not officially documented or supported. Use `np.max(res.logl)` instead.
- Fixed bug that caused non-reproducible behavior in `nest_lc()` even when `numpy.random.seed()` was called directly beforehand. [#102]

- Fixed file I/O problems on Python 3 related to string encoding. [#83, #85]
- Fixed problem with SDSS bandpasses being stored as integers internally, preventing them from being used with models with dust. [#100, #101]
- Fixed problem where built-in source name and version strings were being dropped. [#82]
- Minor doc fixes.

12.1.9 v1.0.0 (2015-02-23)

- **[API change] The API of `mcmc_lc` has changed significantly** (the function was marked experimental in previous release).
- **[Deprecation]** In result of `fit_lc`, `res.cov_names` changed to `res.vparam_names`.
- **[Deprecation]** In result of `nest_lc`, `res.param_names` changed to `res.vparam_names`. This is for compatibility between the results of `fit_lc` and `nest_lc`. [#30]
- **[Deprecation]** Deprecate `flatten` keyword argument in `fit_lc()` in favor of explicit use of `flatten_result()` function.
- Many new built-in models.
- Many new built-in bandpasses.
- New remote data fetching system.
- SALT2 model covariance available via `Model.bandfluxcov()` method and `modelcov=True` keyword argument passed to `fit_lc`.
- New simulation function, `zdist`, generates a distribution of redshifts given a volumetric rate function and cosmology.
- New simulation function, `realize_lcs`, simulates light curve data given a model, parameters, and observations.
- Add color-related keyword arguments to `plot_lc()`.
- Add `tighten_ylim` keyword argument to `plot_lc()`.
- Add `chisq()` function and use internally in `fit_lc()`.
- Add `SFD98Map` class for dealing with SFD (1998) dust maps persistently so that the underlying FITS files are opened only once.
- Update `get_ebv_from_map()` to work with new SkyCoord class in `astropy.coordinates` available in astropy v0.3 onward. Previously, this function did not work with astropy v0.4.x (where older coordinates classes had been removed).
- Update to new configuration system available in astropy v0.4 onward. This makes this release incompatible with astropy versions less than 0.4.
- Now compatible with Python 3.
- Increased test coverage.
- Numerous minor bugfixes.

12.1.10 v0.4.0 (2014-03-26)

This is non-backwards-compatible release, due to changes in the way models are defined. These changes were made after feedback on the initial design.

The most major change is a new central class `Model` used throughout the pacakge. A `Model` instance encompasses a `Source` and zero or more `PropagationEffect` instances. This is so that different source models (e.g., SALT2 or spectral time series models) can be combined with arbitrary dust models. The best way to think about this is `Source` and `PropagationEffect` define the rest-frame behavior of a SN and dust, and a `Model` puts these together to determine the observer-frame behavior.

- New classes:
  - `sncosmo.Model`: new main container class
  - `sncosmo.Source`: replaces existing `Model`
  - `sncosmo.TimeSeriesSource`: replaces existing `TimeSeriesModel`
  - `sncosmo.StretchSource`: replaces existing `StretchModel`
  - `sncosmo.SALT2Source`: replaces existing `SALT2Model`
  - `sncosmo.PropagationEffect`
  - `sncosmo.CCM89Dust`
  - `sncosmo.OD94Dust`
  - `sncosmo.F99Dust`
- New public functions:
  - `sncosmo.read_griddata_ascii`: Read file with `phase wave flux` rows
  - `sncosmo.read_griddata_fits`
  - `sncosmo.write_griddata_fits`
  - `sncosmo.nest_lc`: Nested sampling parameter estimation of SN model
  - `sncosmo.simulate_vol` (EXPERIMENTAL): simulation convenience function.
- Built-ins:
  - updated SALT2 model URLs
  - added SALT2 version 2.4 (Betoule et al 2014)
- Improvements to `sncosmo.plot_lc`: flexibility and layout
- Many bugfixes

12.1.11 v0.3.0 (2013-11-07)

This is a release with mostly bugfixes but a few new features, designed to be backwards compatible with v0.2.0 ahead of API changes coming in the next version.

- New Functions:
  - `sncosmo.get_ebv_from_map`: E(B-V) at given coordinates from SFD map.
  - `sncosmo.read_snana_ascii`: Read SNANA ascii format files.
  - `sncosmo.read_snana_fits`: Read SNANA FITS format files.

  - `sncosmo.read_snana_simlib`: Read SNANA ascii "SIMLIB" files.
- registry is now case-independent. All of the following now work:

```
sncosmo.get_magsystem('AB')
sncosmo.get_magsystem('Ab')
sncsomo.get_magsystem('ab')
```

- Photometric data can be unordered in time. Internally, the data are sorted before being used in fitting and typing.
- Numerous bugfixes.

12.1.12 v0.2.0 (2013-08-20)

- Added SN 2011fe Nearby Supernova Factory data to built-in models as `'2011fe'`
- Previously "experimental" functions now included:
  - `sncosmo.fit_lc` (previously `sncosmo.fit_model`)
  - `sncosmo.read_lc` (previously `sncosmo.readlc`)
  - `sncosmo.write_lc` (previously `sncosmo.writelc`)
  - `sncosmo.plot_lc` (previously `sncosmo.plotlc`)
- New functions:
  - `sncosmo.load_example_data`: Example photometric data.
  - `sncosmo.mcmc_lc`: Markov Chain Monte Carlo parameter estimation.
  - `sncosmo.animate_model`: Model animation using matplotlib.animation.
- Fitting: `sncosmo.fit_lc` now uses the iminuit package for minimization by default. This requires the iminuit package to be installed, but the old minimizer (from scipy) can still be used by setting the keyword `method='l-bfgs-b'`.
- Plotting: Ability to plot model synthetic photometry without observed data, using the syntax:

```
>>> sncosmo.plot_lc(model=model, bands=['band1', 'band2'])
```

- Photometric data format: Photometric data format is now more flexible, allowing various names for table columns.

12.1.13 v0.1.0 (2013-07-15)

Initial release.

12.2 About SNCosmo

12.2.1 Package Features

- **SN models:** Synthesize supernova spectra and photometry from SN models.
- **Fitting and sampling:** Functions for fitting and sampling SN model parameters given photometric light curve data.

- **Dust laws:** Fast implementations of several commonly used extinction laws; can be used to construct SN models that include dust.
- **I/O:** Convenience functions for reading and writing peculiar data formats used in other packages and getting dust values from SFD (1998) maps.
- **Built-in supernova models** such as SALT2, MLCS2k2, Hsiao, Nugent, PSNID, SNANA and Whalen models, as well as a variety of built-in bandpasses and magnitude systems.
- **Extensible:** New models, bandpasses, and magnitude systems can be defined, using an object-oriented interface.

12.2.2 Relation to other SN cosmology software

There are several other publicly available software packages for supernova cosmology. These include (but are not limited to) snfit (SALT fitter), SNANA and SNooPy (or snpy).

- snfit and SNANA both provide functionality overlapping with this package to some extent. The key difference is that these packages provide several (or many) executable applications, but do not provide an API for writing new programs building on the functionality they provide. This package, in contrast, provides no executables; instead it is a *library* of functions and classes designed to provide the building blocks commonly used in many aspects of SN analyses.
- SNooPy (or snpy) is also a Python library for SN analysis, but with a (mostly) different feature set. SNCosmo is based on spectral timeseries models whereas SNooPy is more focussed on models of light curves in given bands.

12.2.3 The name SNCosmo

A natural choice, "snpy", was already taken (SNooPy) so I tried to be a little more descriptive. The package is really specific to supernova *cosmology*, as it doesn't cover other types of supernova science (radiative transfer simulations for instance). Hence "sncosmo".

12.2.4 Contributors

Alphabetical by last name:

- Stephen Bailey
- Kyle Barbary
- Tom Barclay
- Rahul Biswas
- Matt Craig
- Ulrich Feindt
- Brian Friesen
- Danny Goldstein
- Saurabh Jha
- Steve Rodney
- Caroline Sofiatti
- Rollin C. Thomas

- Michael Wood-Vasey

12.3 Contributing

12.3.1 Overview

SNCosmo follows the same general development workflow as astropy and many other open-source software projects. The astropy development workflow page documents the process in some detail. While you should indeed read that page, it can seem a bit overwhelming at first. So, we present here a rough outline of the process, and try to explain the reasoning behind it.

The process is centered around git and GitHub, so you need to know how to use basic git commands and also have a GitHub account. There is a "blessed" copy of the repository at https://github.com/sncosmo/sncosmo. Individual contributors make changes to their own copy (or "fork" or "clone" in git parlance) of the repository, e.g., https://github.com/kbarbary/sncosmo, then ask that their changes be merged into the "blessed" copy via a Pull Request (PR) on GitHub. A maintainer (currently Kyle) will review the changes in the PR, possibly ask for alterations, and then eventually merge the change.

This seems overly complex at first glance, but there are two main benefits to this process: (1) Anyone is free to try out any crazy change they want and share it with the world on their own GitHub account, without affecting the "blessed" repository, and (2) Any proposed changes are reviewed and discussed by at least one person (the maintainer) before being merged in.

12.3.2 Detailed steps

Do once:

1. Hit the "fork" button in the upper right hand corner of the https://github.com/sncosmo/sncosmo page. This creates a clone of the repository on your personal github account.
2. Get it onto your computer (replace username with your GitHub username):

```
git clone git@github.com:username/sncosmo.git
```

3. Add the "blessed" version as a remote:

```
git remote add upstream git@github.com:sncosmo/sncosmo.git
```

   This will allow you to update your version to reflect new changes to the blessed repository that others have made).

4. Check that everything is OK:

```
$ git remote -v
origin    git@github.com:username/sncosmo.git (fetch)
origin    git@github.com:username/sncosmo.git (push)
upstream    git@github.com:sncosmo/sncosmo.git (fetch)
upstream    git@github.com:sncosmo/sncosmo.git (push)
```

   You can call the remotes anything you want. "origin" and "upstream" have no intrinsic meaning for git; they're just nicknames. The astropy documentation calls them "your-github-username" and "astropy" respectively.

Every time you want to make a contribution:

1. Ensure that the clone of the repository on your local machine is up-to-date with the latest upstream changes by doing `git fetch upstream`. This updates your local "remote tracking branch", called `upstream/master`.
2. Create a "topic branch" for the change you want to make. If you plan to make enhancements to the simulation code, name the branch something like "simulation-enhancements":

   ```
   git branch simulation-enhancements upstream/master
   ```

   (`upstream/master` is where the branch branches off from.)
3. Move to the branch you just created:

   ```
   git checkout simulation-enhancements
   ```

4. *Make changes, ensure that they work, etc. Make commits as you go.*
5. Once you're happy with the state of your branch, push it to your GitHub account for the world to see:

   ```
   git push origin simulation-enhancements
   ```

6. Create a PR: Go to your copy on github (https://github.com/username/sncosmo) select the branch you just pushed in the upper left-ish of the page, and hit the green button next to it. (It has a mouse-over "compare, review, create a pull request")

What happens when the upstream branch is updated?

Suppose that you are following the above workflow: you created a topic branch `simulation-enhancements` and made a few commits on that branch. You now want to create a pull request, but there's a problem: while you were working, more commmits were added to the `upstream/master` branch on GitHub. The history of your branch has now diverged from the main development branch! What to do?

1. Fetch the changes made to the upstream branch on so that you can deal with the changes locally:

   ```
   git fetch upstream
   ```

   This will update your local branch `upstream/master` (and any other `upstream` branches) to the match the state of the upstream branch on GitHub. It doesn't do any merging or resolving, it just makes the new changes to `upstream/master` visible locally.
2. There are two options for this next step: `merge` or `rebase` with the latter being preferred for this purpose. Assuming you are on your branch `simulation-enhancements`, you *could* do `git merge upstream/master`. This would create a merge commit that merges the diverged histories back together. This works, but it can end up creating a confusing commit history, particularly if you repeat this process several times while working on your new branch. Instead, you can do:

   ```
   git rebase upstream/master
   ```

   This actually *rewrites* your commits to make it look like they started from where `upstream/master` now is, rather than where it was when you started work on your `simulation-enhancements` branch. Your branch will have the exact same contents as if you had used `git merge`, but the history will be different than it would have been if you had merged. In particular, there is no merge commit created, because the history has been rewritten so that your branch starts where `upstream/master` ends, and there is no divergent history to resolve. This means you can rebase again and again without creating a convoluted history full of merges back and forth between the branches.

Trying out new ideas

git branches are the best way to try out new ideas for code modifications or additions. You don't even have to tell anyone about your bad ideas, since branches are local! They only become world visible when you push them to GitHub. If, after making several commits, you decide that your new branch `simulation-enhancements` sucks, you can just create a new branch starting from upstream/master again. If it is a really terrible idea you never want to see again, you can delete it by doing `git branch -D simulation-enhancements`.

Obviously this isn't a complete guide to git, but hopefully it jump-starts the git learning process.

12.3.3 Developer's documentation: release procedure

These are notes mainly for the one person that manages releases. Yes, this could be more automated, but it isn't done very often, and involves some human verification.

- Update `docs/history.rst` with a summary of the new version's changes.
- Bump version in `setup.py`.
- Check copyright year in `docs/conf.py`.
- Build package and docs and check that docs look good.
- Commit.
- `git clean -dfx`
- `setup.py sdist`
- Check that the tarball in `dist/` can be unpacked and that `setup.py test` succeeds. Bonus: create a fresh conda environment (or virtual environment) with minimal requirements and install and test in that.
- `setup.py register`
- `setup.py sdist upload`

Post-release steps:

- If not a bugfix release, create a feature branch. For example, `git branch v1.1.x`.
- Tag the release. For example, `git tag v1.1.0`.
- On master, bump version in `setup.py` to the next development version and add the next development version to `docs/whatsnew.rst`.
- Commit.
- Push repo changes to GitHub. For example: `git push upstream master v1.1.x v1.1.0`.

Docs and conda

- On readthedocs.org, set the new feature branch to "active".
- To trigger new conda build, edit version number in requirements.txt in https://github.com/astropy/conda-builder-affiliated and submit a pull request.
- Once conda build succeeds, make the new feature branch the default on readthedocs.org.
- Check out the source code: https://github.com/sncosmo/sncosmo
- Report bugs, request features: https://github.com/sncosmo/sncosmo/issues
- User & developer mailing list: https://groups.google.com/forum/#!forum/sncosmo

Bibliography

[N023] Nugent, Kim & Permutter 2002

[S043] Stern, et al. 2004

[L053] Levan et al. 2005

[G993] Gilliland, Nugent & Phillips 1999

[S113] Sako et al. 2011

[H073] Hsiao et al. 2007

[G103] Guy et al. 2010

[B14b3] Betoule et al. 2014

[P133] Pereira et al. 2013

[Whalen133] Whalen et al. 2013

[Jha073] Jha, Riess and Kirshner 2007

[R4] extracted from SNANA's SNDATA_ROOT on 29 March 2013.

[R5] extracted from the SNooPy package on 21 Dec 2012.

[R6] extracted from SNANA's SNDATA_ROOT on 15 August 2013.

[R7] extracted from SNANA's SNDATA_ROOT on 5 August 2014.

[R8] private communication (D.Whalen, May 2014).

[R9] In MLCS2k2 language, this version corresponds to "MLCS2k2 v0.07 rv19-early-smix vectors"

[B900] Bessell 1990, Table 2

[B14a0] Betoule et al. (2014), Footnote 21

[D100] Doi et al. 2010, Table 4

Python Module Index

Index

Symbols

A

B

C

E

F

G

L

M

N

O

P

R

S

T

U

W

Z