# Using SNooPy[*]

Chris Burns

June 29, 2012

## 1 Introduction

The philosophy if SNooPy is to provide both an interactive environment to fit TypeIa supernova (SNIa) light-curves as well as a set of routines that can be called within a scripting environment that would allow the user to write automated fitting routines. The goal is not to lock the user into one kind of method, like stretch, $\Delta m_{15}$ or what have you. Rather, the idea is to provide generic tools and specific components of established methods and let the user decide how best to fit the data. In this way, the software can be thought more of as a laboratory environment rather than a specific program.

Given that this is a very modular way to program, it doesn't lend itself very well to compiled languages. So one has a choice of higher-level scripting languages: python, tcl, perl, or even the IRAF cl. In this case, Python was chosen. There are many reasons for this. First, I have the impression that Python is becoming more prevalent in Astronomy (for example, some of the newer STSci routines in IRAF are only provided through the python implementation of the cl: pyraf). Second, python is designed from the ground up as an object-oriented (OOP) language (unlike perl, tcl, IRAF cl or even IDL). OOP is very well suited for large modular software packages. Lastly, python has access to two very powerful numerical engines: scipy and pyraf. Scipy is a general-purpose package of scientific routines while pyraf gives one access to the IRAF library of routines. Together with a very large array of visualization and database modules, python has everything needed to make this package work. Python is also, in this author's opinion, the easiest and most intuitive programming language to learn and debug. The one thing python lacks is standardization, in the sense that there is an overabundance of modules and solutions to specific problems. In this instance, plotting and numerical arrays (vectors, matrices) have many different solutions. For plotting, I initially chose PGPLOT, as it has a long history in the astronomical community. However, it now appears that matplotlib is winning out as the most standard plotting package for python, so I have integrated it in as well. PGPLOT is still faster and more tested, so I recommend it, but if you prefer matplotlib, you can use it instead (see installation instructions for selecting plotting package). For the arrays, I've chosen Numpy, which his now the de-facto array handling package.

Do you need to know Python to use SNooPy? It depends what you want to do. If you want to interactively fit light-curves, then you really don't need to know how to program in Python. If you want to write a script that automatically fits light-curves or get at the

[*]SuperNovae in object oriented Python.

underlying layers of the code then, yes, you need to learn Python. Even if you don't want to program, knowing a bit of python syntax can really help in the interactive shell. A good place to get started with learning python is the python web page: http://www.python.org `http://www.python.org`.

SNooPy also uses ipython, an extended version of the python interactive shell. This shell has many of its own features and tricks that you might find useful. More information on ipython can be found on its project web page: http://ipython.scipy.org `http://ipython.scipy.org`.

## 2 A Simple Example

First, to start SNooPy, you simply use the command `snpy`. This is a script that starts up ipython and loads in the SNooPy modules.

To give an idea of what the software package looks like and how it behaves, here is a quick session which shows loading the data, fitting a light-curve, plotting the results and saving the parameters.

```
$ snpy
Python 2.5.4 (r254:67916, Aug 26 2009, 16:15:39)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.4 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features. %quickref -> Quick referen
object?   -> Details about 'object'. ?object also works, ?? prints more.

In[1]: s = get_sn('SN2006X.txt')

In[2]: s.fit()

In[3]: s.summary()

------------------------------------------------------------------------------
SN  SN2006X
z = 0.005          ra=185.72500         dec= 15.80920
Data in the following bands: B,  g,  i,  H,  K,  J,  r,  u,  V,  Y,
Fit results (if any):
   Observed B fit to restbad B
   Observed g fit to restbad g
   Observed i fit to restbad i
   Observed H fit to restbad H
   Observed K fit to restbad K
   Observed J fit to restbad J
   Observed r fit to restbad r
   Observed u fit to restbad u
   Observed V fit to restbad V
   Observed Y fit to restbad Y
   EBVhost = 1.518  +/-  0.014
   Tmax = 53786.288  +/-  0.126
   DM = 31.316  +/-  0.024
   dm15 = 1.091  +/-  0.026

In [4]: s.plot(single=1)

[Plot of SN2006X is shown on screen with all filters on a single graph]

In [5]: s.save('SN2006X.snpy')

In [6]:
```

On input line 1, an instance of the `sn` class[1] is created by giving the name of a file containing the SN light-curves in the flat ASCII format (see section 6.2). The data is retrieved from the file and loaded into the instance variable `s`. This object not only contains the photometric data for SN2006X, it also contains the fit parameters. All the fitting, plotting, and utility functions are part of the instance. Notice that in line 2 above, the fit function is part of the `s` object. On line 3, I ask for a summary of the fit, which shows the best-fit values of the parameters and tells us which rest-frame filters were used to fit the observed filters. Note that SNooPy handles the K-corrections (or cross-band K-corrections) as part of the fitting process. On line 4, we make a plot, which shows up on screen (and has buttons to save to a file if you wish). Finally, on line 6, I save the instance to file so that I can load it up at a later time (saving as an instance saves all the fitting parameters, K-corrections, etc. as well as the light-curve data).

There are other classes besides the `sn` class, which can be used inside SNooPy. For example, there is a `filter` class that has data and functions relating to photometric band-passes. There are also a couple of `template` classes that can be used to generate light-curve templates (currently using $\Delta m_{15}$ and $s_{BV}$ as parameters). Here are a couple of examples:

```
In [7] B = fset['B']
In [8] V = fset['V']
In [9] seds = getSED([1,2,3,4,5])
In [10] Bmags = array([B.synthmag(sed) for sed in seds])
In [11] Vmags = array([V.synthmag(sed) for sed in seds])
In [12] t = CSPtemp.template()
In [13] t.mktemplate(1.5)
In [14] BminusV = t.B - t.V
In [15] BminusVsynth = Bmags - Vmags
In [16] deltaBV = BminusV[16:21] - BminusVsynth
In [17] print deltaBV
```

Here, in the first two lines, we've setup some filters (CSP B and V). The structure `fset` is the filter set and contains many different filters (see section 10). Then, we retrieve Eric Hsiao's template for days 1 through 5 after maximum. In lines 10 and 11, we compute synthetic magnitudes for each day and generate a list (this is an example where knowing

[1]This used to be called the `super` class in older versions, but `super` has become a python keyword, so is off-limits now.

python syntax can help a lot). We use the array function to ensure that these are python arrays, which allows us to do math on them later. Line 12 constructs a template instance and line 13 generates a template with $\Delta m_{15} = 1.5$. We can then compare the B-V color obtained with Prieto's light-curve templates and what you get by making synthetic magnitudes from Eric's SED template. Note that the template is defined from day -15, so element 15 is day 0, 16 is day 1, etc.

## 3 Data Structure

Python is object-oriented and SNooPy has been designed[2] with this in mind. As such, the data are organized into a hierarchical structure of objects. The base (or parent) object is the supernova itself (the `sn` class). It contains scalar variables like coordinates, redshift, Milky-Way reddening, etc. It has functions for plotting, fitting, and saving itself (see section 4). It has variables that control the way the light-curves are fit.

The `sn` instance also contains references to a number of light-curves, one for each filter (the `lc` class), which are stored in a python dictionary called `data`. The dictionary is indexed by the observed filter name. These lc objects contain the data (time, magnitude, flux, errors, etc), as well as functions for plotting, interpolation, and other useful tasks that apply to one light-curve at a time. Each `lc` object also has a reference to a `filter` object that defines the filter response, the zero-point, and lots of other goodies. There is also a shortcut for accessing the `lc` instances: you can refer to them as member variables. So `s.data['B']` and `s.B` are equivalent.

Lastly, the sn instance also has an object that defines the model that will be used to fit the data. It holds the parameter values, errors, $\chi^2$, etc. It also does all the heavy lifting when fitting the model to the data. Because the model is an abstract object, it can be replaced with other model objects transparently (so long as the replacement model object conforms to the structure expected by SNooPy; see section 8).

The outline of the major objects is show in figure 1. This is not exhaustive

## 4 Data Persistence

The current version of `SNooPy` gets and saves its data from three possible sources: a `mySQL` database, a file that was created using the `sn.save()` function, or a flat text file of the proper format. A single convenience function, `get_sn()` can be used to load a sn object from any of these sources.

For large datasets, I highly recommend using the mySQL solution: it allows for data persistence that is kept separate from one user/computer combination. The major downside is you have to install mySQL (not too bad) and setup a schema which matches what SNooPy expects (still not too bad) and possibly learning mySQL in the first place (well, you always *meant* to learn it, didn't you? This is your excuse). I'm not going to explain how to install mySQL and setup a schema, that's best left for the SQL documentation. The required schema is detailed in Appendix **??**.

If you don't want the bother of SQL, then you can load initial data from a flat ascii data file (see section 6.2). When you're done fitting and want to save the sn object, with all its

[2]If you can call this cobbled-together-late-at-night-while-observing code a design.

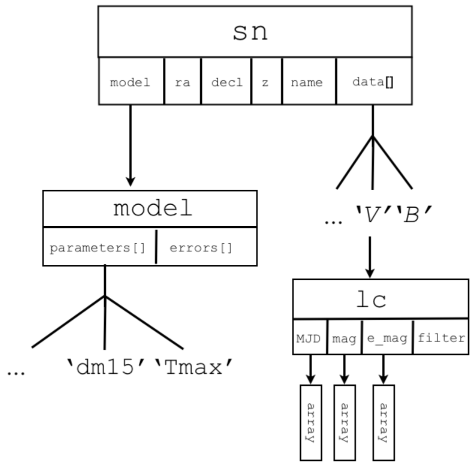

Figure 1: Object structure in SNooPy.

state variables, simply use the `sn.save('filename')` function to save the `sn` instance as a python pickle file. You can later load it in using `get_sn('filename')`.

WARNING: because the pickle file will contain the entire data structure of the SN object, there is no guarantee that pickle files will be loadable by different versions of SNooPy. In fact, I can pretty-much guarantee that major upgrades of SNooPy will result in non-compatibility in pickled files.

## 5 Fitting Light-curves

All the light-curve fitting is done through a member function of the `sn` class: `fit()`. You basically tell `fit()` which filters to fit (or all by default) and which parameters to hold constant. This function sets up some initial book-keeping, and then hands off the work to a `model` instance. The `model` class defines the model that will be used to fit the light-curves. I did it this way so that adding new models (or modifying existing ones) would be easier. SNooPy comes with three built-in models: `EBV_model, EBV_model2,` and `max_model`, which I will explain in the next two sections. At any time, you can switch between the two by using the `choose_model()` member function of the `sn` class (see section 6.3.4). As of SNooPy version 2, you can fit the `EBV_model2` and `max_model` using either $\Delta m_{15}$ or a new stretch-like parameter $s_{BV}$ which is introduced in the CSP's most recent analysis paper (Burns et al., (in prep)).

### 5.1 EBV_model

This light-curve model is a variation of that given by **?**. We compare the observed data in $N$ filters to the following model:

$$\begin{aligned} m_X\left(t - t_{max}\right) &= T_Y\left(\left(t' - t_{max}\right)/(1+z), \Delta m_{15}\right) + M_Y\left(\Delta m_{15}\right) + \mu + R_X E\left(B-V\right)_{gal} + \\ & R_Y E\left(B-V\right)_{host} + K_{X,Y}\left(z, \left(t' - t_{max}\right)/(1+z), E\left(B-V\right)_{host}, E\left(B-V\right)_{gal}\right) \end{aligned}$$

where $m_X$ is the observed magnitude in band $X$, $t_{max}$ is the time of B maximum, $\Delta m_{15}$ is the decline rate parameter (?), $M_Y$ is the absolute magnitude in filter $Y$ in the rest-frame of the supernova, $E(B-V)_{gal}$ and $E(B-V)_{host}$ are the reddening due to galactic foreground and host galaxy, respectively, $R_X$ and $R_Y$ are the total-to-selective absorptions for filters $X$ and $Y$, respectively, $K_{XY}$ is the cross-band k-correction from rest-frame $X$ to observed filter $Y$. Note that the k-corrections depend on the epoch and *can* depend on the host and galaxy extinction (as these modify the shape of the spectral template). In the fitting, one has the choice of the spectral template of ? or ?. The latter is the default. You can choose to allow the K-corrections to vary during the fit, keep them fixed, or not use them at all. See the parameters for the `fit()` function in section 6.3.2.

The template $T(t, \Delta m_{15})$ can be generated from the code of ? or ?. In the former case, you will be fitting to rest-frame $B_sV_sR_sI_s$[3] while in the latter case, you can fit to $uBVgriYJH$ from the CSP data (?). You can mix and match which template generator you use: it is all a matter of which filter you choose for the `restbands` instance variable (see section 5.5). In all, this model fits 4 parameters: `Tmax`, `dm15`, `EBVhost`, and `DM`.

Note that to determine the host reddening, you need to fit at least two distinct *restframe* filters. For now, I've left the galactic and host reddening laws ($R_X$ and $R_Y$) as member variables rather than parameters to be fit[4]. This could change in the future.

## 5.2 EBV_model2

This is the same model as EBV_model, except that it can fit both $\Delta m_{15}$ - and $s_{BV}$-based templates. And instead of the calibration presented in ?, the calibration from the upcoming CSP analysis paper (Burns et al, in prep) is used. When using the `choose_model` function to select this model, you can specify `stype='dm15'` or `stype='st'` to select the two different light-curve parameters.

## 5.3 max_model

An alternative to assuming some kind of relationship between the different filters (i.e., some kind of reddening law as is done in the `EBV_model`), one can simply fit the maximum magnitude of each light-curve. This is how the `max_model` model works. It uses the same light-curve templates as `EBV_model`, but instead fits the following model:

$$\begin{aligned} m_X(t - t_{max}) &= T_Y\left((t' - t_{max})/(1+z), \Delta m_{15}\right) + m_Y + R_X E(B-V)_{gal} + \\ &\quad K_{X,Y}\left(z, (t'-t_{max})/(1+z), E(B-V)_{host}, E(B-V)_{gal}\right) \end{aligned}$$

where $m_Y$, the peak magnitude in filter $Y$is now a parameter. Of course, depending on the number of filters you fit, you will have that many $m_Y$ parameters. Note that K-corrections and Milky-way extinction are performed exactly as in the `EBV_model` case. Therefore, for $N$ filters, max_model will have $N+2$ parameters: `Tmax`, `dm15`, and $N$ $f$`max`, where $f$ is the rest-band filter name (so if, for instance, you fit an observed light-curve to restframe $B$, there will be a `Bmax` parameter). As with the `EBV_model2` model, you can choose which light-curve parameter you wish to use by specifying the `stype` argument to `choose_model`().

[3]The 's' subscript refers to 'standard', which is to say the Bessel filters from ?

[4]So far, data I've analyzed hasn't been good enough to distinguish between reddening laws.

**NOTE:** Please be aware that any non-linear fitter will only find a *local* minimum in $\chi^2$. It is up to you, the user, to try different starting points in parameter space and see which one gives the overall best-fit. After each fit, the member variable `rchisq` is updated with the reduced-$\chi^2$. You can therefore use this to discriminate between different solutions.

## 5.4 K-corrections and Reddening coefficients

Regardless of the method used, there are two generic problems when fitting light-curves. First, because you are observing a SN that is at some redshift, the observed spectral energy distribution (SED) is shifted to the red compared to what would be observed at zero redshift. As as a result, the filters are sampling a bluer part of the SNIa spectrum. K-corrections are needed to take this into account and we therefore need either observed spectra of the SN or some kind of SED template to make this calculation. We also need to take into account, if possible, of any factors that might alter the shape of the SED (like reddening). Once the SED is known, it is straightforward to compute the K-correction (see **??**).

Second, because the SED of a SNIa is significantly different from a stellar SED and also varies with time, the ratio of selective to total absorption ($R_X$) is going to depend on the redshift of the SN, the epoch of observation, and the amount of host reddening (and any other factor that modifies the intrinsic SED). Once the SED is known (or assumed), the reddening coefficient $R_X$ can be deduced.

The basic problem here is that we need to know the observed SED as well as possible. Currently, SNooPy simply takes the SED of **?** or **?** (user's choice), as the SED. This is used to compute initial k-corrections that are simply a function of time. If reddening information is known (from a fit value of EBVhost, for instance), the reddening law of **?**, further modified by **?** is applied to the SED before computing the K-correction or reddening coefficient. After this initial fit, the $N-1$ colors of the SN are computed and these can be used to warp the SED so that synthetic colors obtained from the SED match the observed colors form the fit (see **?** ). The hope is that this "warped" SED is the best guess at the correct SED. This step in the fit process may be skipped by using the `mangle=0` argument to the `fit()` function.

## 5.5 Doing the Fit

The `fit()` function is a member of the `sn` class. The first argument is a list of observed filters to fit (or by default, fit them all). Following this argument, you can specify values for any of the parameters, which will keep them fixed during the iteration: `Tmax, dm15, s, EBVhost, DM`, and any of the `fmax`. By default, the fitter will try to fit the observed filter set with the same rest-band filters. It is therefore important that your filter names match the filters provided by the templates: `Bs,Vs,Rs,Is` (for **?** templates) or `u,B,V,g,r,i,Y,J,H` (for CSP templates). If your observed filter does not match any of these (either because you are working at high-z and want to do a cross-band K-correction, or are working in a different photometric system), you need to specify the `restbands` for each observed filter (`restbands` is a member variable that acts like a python dictionary and maps observed filters to rest filters). Here is a short example:

```
In [1] s = get_sn('04D1oh')
In [2] s.restbands['g_m'] = 'B'   ;#  Fit observed g_m data with 'B' (CSP)template
In [3] s.fit(['g_m'], EBVhost=0, dm15=1.1)
In [4] s.fit(['g_m'], EBVhost=0)
In [5] s.restbands['r_m'] = 'V';  s.restbands['i_m'] = 'r';  s.restbands['Jc'] = 'i'
In [6] s.fit(['g_m','r_m','i_m','Jc'], dm15=s.dm15, Tmax=s.Tmax)
In [7] s.fit(['g_m','r_m','i_m','Jc'])
In [8] save(s, "my_lovely_fit.snpy")
```

On line 1, we make a new instance. On line 3, we decide to fit the megacam g filter with a rest-frame CSP B template. On line 3, we fit only the g_m filter and therefore restrict `EBVhost` to 0 (since we don't have any color information). We can also fix $\Delta m_{15} = 1.1$ for a first attempt at a fit (this can help if you have pretty low S/N data). On line 4, we let $\Delta m_{15}$ go free. On line 5, we specify more rest-band filters. On line 6 we add more filters to the fit and allow the host extinction to vary, though keeping $\Delta m_{15}$ and $t_{max}$ fixed at the values determined from the previous fit (all fitted parameters and their errors are saved as member data of the `sn` instance). On line 7, we allow all parameters to vary. On line 8, we save the fit to a file that can be later loaded into SNooPy (using something like `s=load(filename)`).

The fit is performed using the Lavenberg-Marquardt algorithm for minimizing $\chi^2$. Unless you specify otherwise, all the fitting is done in flux units (SNooPy takes care of converting magnitudes to fluxes, if needed, for both the observations and models). Note that because the Lavenberg-Marquardt algorithm is an non-linear least-squares solving routine, it is not guaranteed to find the global minimum $\chi^2$. It is your responsibility to inspect the fit to make sure it got it right.

Now that you have seen the basic workings, we can now move on to the details of each class so that you can build your own routines inside SNooPy, or write python scripts that use these classes to fit light-curves.

## 5.6 Getting Help

Python has an internal help system which utilizes comments at the beginning of functions and classes (so-called docstrings). Simply use the built-in help() function to get help on an item. Here are some examples (output is not shown to save space):

```
In [1] help(sn)
In [2] help(sn.fit)
In [3] help(sn.plot)
In [4] help(lc)
```

Line 1 gets help about the entire `sn` class, which will list all the functions defined therein, including internal ones that are not meant to be used by end users (but of course are available to be hacked, but may lack good documentation). Lines 2 and 3 get more specific help on individual member functions. Line 4 gets help on the `lc` (light-curve) class. You can ask for help on any python object (including variables).

# 6 The sn Object

## 6.1 Constructor and its options

Supernova objects are created using the `sn` constructor or using the `get_sn()` convenience function:

```
s = sn(name, z=None, ra=None, dec=None)
s = get_sn('my_favorite_SN.snpy')
```

The optional arguments `z`, `ra`, and `dec` are used if the supernova does not exist in the sql database. The redshift `z` is required to do the fitting (obviously) and `ra`, `dec` are needed to compute the galactic extinction from the Schlegel maps.

## 6.2 Creating an object from a flat file

If you don't have or want an SQL database, you can load in data from a flat file. To do this, simply create a text-file with the following format:

```
{name} {z} {ra} {decl}
filter {filter1}
{Date1} {magnitude1} {error1}
{Date2} {magnitude2} {error2}
...
{DateN} {magnitudeN} {errorN}
filter {filter2}
...
```

The items in {}'s are to be filled in. The first line gives the name, redshift, and right-ascension and declination in decimal degrees. Next, you specify the name of a filter, followed by data points, one per line. Then you can input another filter and so on. The filter names have to be recognized by SNooPy (use `fset.list_filters()` to list them). See section 10.3 for instructions on how to add custom filters. Once you've made this file, you simply use `get_sn()` to make it into a `sn` object.

```
In[15]:  SN = get_sn('name_of_the_file.dat')
\end{verbatime}



\subsection{Member variables}

The \texttt{sn} object has a number of member variables that do one
of three things: 1) contain data, 2) define a LC fit, or 3) modify
the instance's behavior. Each of these is explained in the following
sections. A member variable is accessed as \texttt{instance.variable}.
For instance:

\begin{verbatim}
In[20]:  SN = sn('SN3241')
<output supressed>
In[21]:  Tmax = SN.Tmax
In[22]:  first_B_epoch = SN.B.MJD[0] - Tmax
In[23]:  Max_B_obs = max(SN.B.mag)
```

| Variable | type | Description |
|---|---|---|
| `z` | float | Heliocentric redshift of the Supernova[5] |
| `ra` | float | Right Ascension in decimal degrees |
| `decl` | float | Declination in decimal degrees |
| `data` | dictionary | Holds the light-curve instances. E.g., `s.data['B']` is the B-band LC data instance. See lc class below. |
| `model` | model | The model instance used to do the fitting (see section 8). This can be changed either by hand or by using `self.choose_model()` function (see section 6.3.4). |
| `bands` | list | A list of strings corresponding to the observed bands defined in `data`. |
| *filter* | lc | The LC data instance for band *filter*. For example, `s.B` is the same as `s.data['B']`. |
| `EBVgal` | float | The Milky-way $E(B-V)$ from the Schlegel maps. |
| `e_EBVgal` | float | Error in `EBVgal`. |
| `ks` | dictionary | A dictionary of computed k-corrections. The index is the filter name, the value is an array of k-corrections, one for each observed epoch. |
| *parameter* | float | Any of the parameters from the model. For example, `s.dm15` is equivalent to `s.model.parameters['dm15']` |
| e_*parameter* | float | Error in any of the model parameters. For example, s.e_Tmax is equivalent to s.model.errors['Tmax'] |

Table 1: Data member variables

### 6.2.1 Data

A `sn` instance has a number of member variables that are not meant to be modified directly: they hold data about the SN or the light-curve data. That doesn't mean you *can't* modify it, but my philosophy has been to leave the observed data alone and build everything into the model. The following table lists these variables and a short explanation:

Note that some of the member data are themselves instances of other classes, most notably the `lc` (light-curve) class and `model` class, which are covered in sections 7 and 8, respectively.

### 6.2.2 Proxy Variables

A number of variables that belong to other classes/structures can be accessed directly from the sn instance. For instance, you could refer to the B `lc` instance in two ways: `s.B` or `s.data['B']`. You can also refer to the parameters `Tmax` of the current model in the following three ways: `s.Tmax` or `s.model.Tmax` or `s.model.parameters['Tmax']`. The last cases are the 'real' locations; the others are simply re-directs that were added for convenience. The re-directs are listed in table 1 in italics.

### 6.2.3 Behavior-Modifying Variables

The member variables that modify how the instance behaves are listed in table 2. They are mostly to do with how the data are fit and plotted.

| Name | type | Description |
|---|---|---|
| filter_order | list | A list of strings corresponding to the order in which the filters should be plotted. Also useful if you want to prevent data from being plotted (simply omit the filter) |
| xrange,yrange | list | X and Y range to plot. Example: `s.xrange = [-20,100]` |
| Rv_gal | float | Milky Way reddening law (default: 3.1) |
| fit_mag | boolean | If true (1), fit in magnitude space, otherwise (0), fit in flux space. Default: 0 |
| restbands | dictionary | A dictionary-like object, indexed by observed band with value corresponding rest-band. Used to specify which template is fit to each observed filter. Example: `s.restbands['u'] = 'B'` means fit observed band u with B-band template. If not explicitly set, it is assumed that a restband with the same name as the observed band is used. |
| replot | boolean | Replot the LC's after fitting or change in parameter? Default: 1 |
| quiet | boolean | Keep it quiet (non-verbose output)? Default: 1 |
| k_version | string | Which SED template to use: 'H3' for Eric Hsiao's latest, 'N' for Peter Nugent, '91bg' for Peter Nugent's 91bg SED. Default: H3, unless $\Delta m_{15} > 1.7$, in which case '91bg' is used. |

Table 2: Behavior modifying variables.

## 6.3 Member Functions

These are the functions that you will use to fit light-curves, get information about the fit, plot the data, etc. Each function has explicit arguments (some mandatory, some optional). Also, some of the member variables will alter how a function works (see section 6.2.3). The most important functions are listed first, each with its own subsection, then a final section has the less important, but useful functions. Some functions are not listed as they are internal to the class. Read the code if you want details about any of the inner workings.

### 6.3.1 Plot

`plot(xrange=None, yrange=None, title=None, single=0, dm=1, fsize=1.0, linewidth=3, symbols=None, colors=None, relative=0, legend=1, mask=0, label_bad=1)`

This function simply plots the data to the screen (or other PGPLOT device if requested). All the arguments are optional and change the behavior of the plot. You can specify `xrange` and `yrange` to modify the extent of the plot. You can output to postscript file by specifying `device=■somefile.ps/CPS■` (see your local PGPLOT documentation for what devices are available). You can give the plot a title by providing a string to that argument, change the line width and default symbol size with `linewidth` and `fsize`, respectively.

If you specify `single=1`, then all the filters are plotted on the same graph, with a magnitude offset of `dm` between them.

You can modify the colors and symbols used for each filter by passing a dictionary of filter-value pairs. For example, `colors={'B':'blue', 'R':'red', 'I':'orange'}` and `symbols={'B':1, 'R':2, 'I':3}`. See the `matplotlib` documentation for the number-symbol combinations or matplotlib documentation for symbols and colors.

### 6.3.2 Fit

`fit(bands=None, {parameter values}, kcorr=1, mangle=1, method='tspline')`

This function is the heart of the software: fitting a light-curve to the data by minimizing $\chi^2$. If you wish to restrict which filters are fit, specify `bands`, a list of filters to fit, otherwise all filters are fit. So if, for example, you had data in B, V, r' (Sloan r), Jc, and Yc, you would specify ['B','V','r_s','Jc','Yc'] as the first argument. This will simultaneously fit the data in these filters to templates specified in the rest-bands member variable. So, if `restbands`={'B':'B', 'V':V', 'r_s':'R', 'Jc':'I', 'Yc':'I'}, then the data in B would be fit with a B template, V with a V template, r_s with an R template, Jc with an I template and Yc with an I template (not that you'd really want to do this).

The arguments {parameter values} are where you assign values to parameters of the model in order to keep them fixed. If a parameter is not specified, it is free to vary. The current values are always used as starting points. Consult the model documentation to find out what parameters there are.

The remaining arguments are:

- `kcorr`: If true, compute k-corrections after an initial fit to find the time of maximum, then fit again. If you want more control over how this is done, set `kcorr=0`, run `kcorr()` manually, then re-fit again with `kcorr=0`.
- `mangle`: If true, the current model of the photometry is used to construct colors as a function of time. The SNIa SED is "mangled" to match these colors before the k-corrections are computed.
- method: If mangling, then the functional form to use when mangling. Currently, a tension spline ('`tspline`') and Cardelli law ('`ccm`') are the two options. '`ccm`' is safer (smoother, fewer parameters), but not as versatile as '`tspline`'.

### 6.3.3 Making k-corrections by hand

```
kcorr(bands=None, mbands=None, mangle=1, interp=1, use_model=0, min_filter_sep=400,
**mopts))
```

This function allows the user to compute k-corrections that are decoupled from the fitting procedure (you might want to do this to have more control over the k-corrections or to play around with the arguments without re-fitting each time). At its simplest (mangle=0), simply use the SNIa SEDs and filter functions to compute k-corrections. If mangle=1, then there are more options.

The idea is that regardless of what has altered the shape of the supernova's SED (extinction or intrinsic variation), the observed colors of the supernova give one a constraint on the overall "tilt" of the spectrum. In the case of many bands, you can actually solve for a higher-order fit (cubic spline, etc).

Simply call `kcorr` and supply it with a list of N filters, from which the N-1 colors will be constructed. These colors, and the filter pass-bands, will be used to find a spline that, when multiplied with the SED, produces the observed colors. By default, these N-1 colors are constructed from bands, but if you want to use only a subset, then specify them in mbands.

To properly estimate the colors for any given day, one needs to have a model for the light-curves. If you choose to use a template, then fit one beforehand. Otherwise, use the fit_spline() function to fit a spline (see section 7.2). If you do nothing, GLoEs will be used to interpolate data. Probably not a good idea at high redshift, where S/N is low. In any case, if you wish to use the real data for the colors where possible, then set interp=0.

When using `mangle=1`, if you have two filters which have very similar effective wavelengths, but different shapes (V and g, for instance), the splines can become very badly behaved. If min_filter_sep is set, then any filter whose effective wavelength is closer than min_filter_sep to another is removed automatically from the set of colors.

### 6.3.4 Other Useful functions

Here is a list and brief summary of other functions belonging to the sn class:

- `choose_model(model)`: Choose which model to use. As of now, this is either "`EBV_model`" or "`max_model`".

- `closest_band(wav, tempbands=['B', 'V', 'R', 'I'])`: For each filter in the data, find the closest rest-frame filter. Returns one of the strings listed in `tempbands`. These strings must represent a valid filter found in the `filters` dictionary (see .

- `compute_w(band1, band2, band3)`: Returns the reddening-free magnitude in the sense that: w = band1 - R(band1,band2,band3)*(band2 - band3) for for instance compute_w(V,B,V) would give: w = V - Rv(B-V). This is still in development.

- `getEBVgal(self)`: Gets the value of E(B-V) due to galactic extinction. The ra and decl member variables must be set beforehand.

- `get_color(band1, band2, nointerp=0)`: return the observed SN color of band1 - band2. Returns a 4-tuple: (MJD, band1-band2, e_band1-band2, flag). Flag is one of: 0 - both bands measured at given epoch; 1 - only one band measured, other interpolated; 2 - extrapolation (based on template) needed; 3 - data interpolated or extrapolated beyond template's definition, so not safe to use!

- `get_mag_table(bands=None)`: This routine returns a table of the photometry, where the data from different filters are grouped according to day of observation. When data is missing, a value of 99.9 is inserted. Format of table is "MJD mag1 emag1 mag2 emag2 ...

- `get_max(bands, restframe=0, deredden=0)`: After a model or spline has been fit, determine the maximum magnitude, error, and time of maximum for the light-curves specified in `bands`. If `rest frame=1`, then remove the K-corrections, thereby converting to rest-frame filters. If `deredden=1`, then remove the galactic and host extinction. Otherwise, the maximum of the model array is used which, due to roundoff, K-corrections, etc. may not be the true maximum. The function returns a 4-tuple: (`maxes`, `e_maxes`, `T_maxes`, `restband`). `maxes` is an array of maxima, `e_maxes` an array of their errors, `T_maxes` an array of time of maxima, and `restband` is a list of the rest-bands used to fit each filter.

- `get_rest_max(bands, deredden=0)`: Simply calls `get_max()` with `restframe=1`, for backward-compatibility.

- `get_restbands()`: Automatically populates the rest-bands member data with filters from the `restbands` member list, whichever effective wavelength is closest to the observed bands. This is run automatically when the sn object is created.
- `load(dictionary)`: Given a previously saved dictionary (as returned by save()), reload the parameters.
- `lira(Bband, Vband, interpolate=0, tmin=30, tmax=90, plot=0)`: Use the Lira law to estimate the extinction. The B-V color is constructed as a function of time using filters `Bband` and `Vband`. The color excess is then estimated to be the median offset between (B-V) and the Lira Law in the time window `tmin < t < tmax`. If `interpolate=1`, then missing data is interpolated. Use `tmin` and `tmax` to restrict which data are used. Use `plot=1` to get a graph. The function returns a 3-tuple: E(B-V), the error and the fitted slope, which can be used as a diagnostic.
- `mask_data()`: Interactively mask out bad data and unmask the data as well. The only two bindings are ”A” (click): mask the data and ”u” to unmask the data.
- `save()`: This will return the parameters dictionary, which can be used to save the state of the fit. Use load() to re-load the parameters.
- `summary()`: Get a quick summary of the data for this SN, along with fitted parameters (if such exist).
- `update_sql(attributes=None, dokcorr=1)`: Updates the current information in the SQL database, creating a new SN if needed. If attributes are specified (as a list of strings), then only these attributes are updated.

## 7 Light-curve class

The light-curve class, `lc`, is a simpler class than sn (for now). It is available for scripting by importing the `lc` module. It basically contains the data of a single filter and a few functions to work with the data. Of particular interest might be the light-curve fitting functions, which allow one to make templates from well-sampled and high S/N data. Depending on the version of `numpy` and if you have `pymc` installed, you can fit light-curves with splines, polynomials, and Gaussian Processes (if you have `pymc` installed).

### 7.1 Member data

Table 3 lists the member variables of the `lc` class.

### 7.2 Member Functions

The most useful member functions are given below:

- `eval(self, times, recompute=0, t_tol=0.1, **args)`: Interpolate (if required) the data to time ’times’. If recompute=1, force a re-computation of the spline coefficients (you can also use any of the arguments for mkspline() here). If there is a data point closer than t_tol away from a requested time, that value is used without interpolation.

| name | type | description |
|---|---|---|
| band | string | name of the filter that this instance represents |
| parent | sn inst. | A pointer to the `sn` instance that contains this `lc` instance. |
| filter | filter inst. | Instance of the filter object that corresponds to self.band |
| MJD | float array | Array of observations dates, usually in Modified Julian Day |
| t | float array | If Tmax has been solved, then this is an array of epochs |
| magnitude | float array | Array of observed magnitudes. |
| e_mag | float array | Array of uncertainties in the magnitudes. |
| K | float array | Array of k-corrections. |
| mag | float array | Like magnitude, but if K are defined, then this returns the k-corrected magnitudes (self.magnitude - self.K). Otherwise, it is equivalent to self.magnitude. |
| flux | float array | Automatically generated array of fluxed, based on the magnitudes and zero point defined by band. |
| e_flux | float array | Array of uncertainties in the flux. |
| tck | list | 3-element list defining a spline: array of knot points, array of spline coefficients, and the order of the spline. This can be used with scipy.integrate.splev() to evaluate the spline at any point (see help page for splev). |
| model | float array | A model of the light-curve based on a template fit (generated automatically by template()). |
| model_t | float array | The times for the model. |
| model_sigmas | float array | Errors in the model. Generated if do_sigma=1 in the call to template() |
| dm15, e_dm15 | floats | The computed value of dm15 and its error (if do_sigma=1) from the spline. Generated by template() |
| Tmax,e_Tmax | floats | The computed value of Tmax and its error (if do_sigma=1) from the spline. Generated by template() |
| Mmax, e_Mmax | floats | The computed value of the maximum magnitude and its error (if do_sigma=1) from the spline. Generated by template() |
| mask | int array | The mask defines which data are good. If mask[i] = 1, then the data at element i is considered good (and will be used in a fit), otherwise, if it is 0, the data is bad and ignored by fits. |

Table 3: Member variables of the lc class

- `mask_emag(self, max)`: Update the lc's mask to only include data with e_mag < max.
- `mask_epoch(self, tmin, tmax)`: Update the lc's mask to only include data between tmin and tmax.
- `template(self, fitflux=False, do_sigma=True, Nboot=50, method=default_method, compute_params=True, **args)`: Generate a smooth interpolation template of the data. You can choose which method to use by specifying method. To get a list of methods available for your setup, use the `list_types()` function. If you wish to fit the flux domain, specify fitflux=True. If you set `compute_params=True`, the interpolator will attempt to measure the following light-curve characteristics and save them as member variables. In order to compute errors in these values, the algorithm may need to do bootstrap errors, in which case, `Nboot` is the number of iterations to use. The computed variables are:
  - `self.Tmax` and `self.e_Tmax`: The time of earliest maximum for the light-curve, with error.
  - `self.Mmax` and `self.e_Mmax`: The magnitude at self.Tmax, with error
  - `self.dm15` and `self.e_dm15`: The Phillips parameter (change in magnitude between Tmax and day 15 in the rest frame of the SN), with error.

  In the following section, I describe the different interpolation methods and their parameters, which you can specify in the call to `template()`.
- `plot(self, flux=0)`: Plot the light-curve and possibly the fitting spline. If an interpolatiing template has been fit, a second panel will show the residuals.

Note: If you fit a template and then use the parent sn instance's plot() function, the spline will be plotted on top of the data (unless there is already a model defined from a fit(), which takes precedence).

## 7.3 The Art of Interpolating

### 7.3.1 Gaussian Processes

If you have installed the pymc package, it comes with an interpolating package that uses Gaussian Processes (GP). GP's are fantastic for interpolating when you may have missing data (gaps). This is usually where other fitters like Dierckx splines and hypersplines (see next sections) go awry.

I'm not going to explain everythign about GP's, but suffice it to say that GP's are like "fuzzy functions". They have a mean value (mean function) and a error (covariance function). In SNooPy's implementation, we employ the Matern covariance function, which has 3 parameters: `scale`, `amp` (amplitude), and `diff_degree` (degree if differentiability). These are conceptually very easy to understand: scale is the scale over which the function typically varies (on the x-axis). Amplitude is the amount by which the function varies on these scales. The degree of differentiability is the "smoothness" of the function. These are the only 3 parameters you need to specify to the template() function to get a fit (or don't specify them at all and take the defaults, which are pretty good for light-curves). If the fucntion does not capture the full behaviour of the light-curve (too smooth), try decreasing the scale and increase the amplitude. If it captures too much of the "noise", increase the scale, lower the amplitde and/or increase the diff_degree.

Here is an example:

```
s.B.template(method='gp', scale=10, amp=s.B.mag.std(), diff_degree=3)
```

The downside to GP's is that they are slower to solve than the other interpolation methods. But I feel that the error estimates that come out (especially in the regions with gaps) are much more robust.

### 7.3.2 Dierckx splines

Splines are a great way to represent data, but they have one very serious drawback: a variable number of parameters. Not only do you have to specify the order of the spline (usually one chooses k=3 for a cubic spline), you also have to choose how many knot points to use. Each knot point has two parameters: its placement on the time axis and the coefficient of the spline at that knot point. You can therefore have the number of knot points equal to the number of data points (plus 2k end-point knots to properly define the spline on the boundaries) and get a spline that's guaranteed to pass through each and every point (an interpolating spline). But real data has noise, so this isn't what you want. At the other extreme, you could use the bare minimum of 2k+2 knots which would give the smoothest spline, but then you lose any interesting structure in the light-curve.

Luckily, `scipy` makes many of the algorithms from Paul Dierckx's book "Curve and Surface Fitting with Splines" (**?**) available. These algorithms deal with automatic choice of the number and placement of the spline knots. They distill the problem down to one parameter: the smoothing, `s`. The larger `s`, the smoother (fewer knots) the curve and the smaller, the more the spline will approach an interpolating spline (s=0). If the weights you provide the spline fitter are proper 1-sigma errors, then setting s to the number of data points should result in a curve with reduced $\chi^2_\nu \simeq 1$, which is what we really want. The first time you fit the spline, set task=0 and the routine will choose an initial set of knot points. If you wish to play around with the smoothing parameter, but keep the knots fixed, use task=1; otherwise, new knots will be chosen each time.

It may be that the resulting spline doesn't "look right". Most likely, this is due to badly estimated weights (variances). The solution is to vary s until you get something that looks right, but of course, this is purely subjective. With proper 1-sigma errors, one can legitimately play around with s in the range $N - \sqrt{2N} < s < N + \sqrt{2N}$ until the spline looks acceptable[6].

Another way to spline is to impose what you think are sensible knots. In this case, simply specify them as an array with the knots argument to the `template()` function and set `task=-1`. In this case, the smoothing is ignored and you get the least-squares spline using these knots. For example:

```
s.B.template(method='spline', task=0, s=len(s.B.mag))
s.B.template(method='spline', task=1, s=len(s.B.mag) - sqrt(2*len(s.B.mag)))
```

[6]I tend to shy away from this, preferring to remove the subjectivity. Rather, I would re-examine the weights and decide whether they are correct. Playing around with s is effectively like globally increasing the errors.

### 7.3.3 Hyperspline

The problem of choosing the correct value of s in the previous section can be removed by using another set of spline routines developed by **?**. Rather than using $\chi^2$ as a statistic for determining the best-fit spline, they use a statistic call the Durbin-Watson statistic. Without getting into the details, this statistic relies more on the auto-correlation in the residuals, which to first order is insensitive to the individual errors of the points. In essence, you're using the pattern of the points to tell you what is noise and what is a true trend.

This spline method, when it works, is the most automatic of the interpolation methods, requiring no parameters (at least initially). Of course, the Achilles-heal of this method is the presence of correlated errors among the data points. But this is likely also a problem with Dierckx splines.

Another case where hyperspline has problems is when there are large gaps in the data. Here, you might want to split up the problem in to two (or more) segments and work on them individually. I hope to add this feature eventually. The parameter you'll most likely need to play with is `lopt` (set the initial number of knots). Set it higher if you find the spline is too smooth.

### 7.3.4 Polynomials

If you have a sufficiently recent version of numpy, you'll have access to several types of polynomials: `polynomial`, `chebyshev`, `laguerre`, `hermite`, and `hermiteE`. Simply use any of these as the method argument to template(). The polynomials have only one free parameter: `n`, the order of the polynomial. For example to fit a Chebyshev polynomial:

```
s.B.template(method='chebyshev', n=5)
```

## 8 Model class

This class is designed to do all the work of fitting a model to data. It uses the Levenberg-Marquardt least-squares algorithm for find the minimum $\chi^2$. It is designed to be sub-classed in order to create specific models. These sub-classes will inherit the basic fitting code and so you can concentrate on setting up the model. In this section, I outline the basic structure of the class, so that you can access the underlying machinery to do your own coding (or design your own model). The casual user need not worry about any of this. SNooPy comes with two models: `EBV_model` and `max_model`.

If you want to create your own model, your best bet is to simply modify the existing `model.py` module and add your own. Simply create a subclass based on the `model` class. You then override `self.parameters`, `self.__init__()`, `self.guess()`, `self.setup()`, `self.__call__()`, and `self.get_max()`. Each is described in a separate section below.

### 8.1 self.parameters

The first thing to define in the model class is the `self.parameters` member variable. This is a dictionary that contains parameter:value pairs. The model will use these parameters to build the numerical model, which is sent to scipy's `leastsq` routine. Another member variable that has the same keys is `self.errors`. This dictionary will have the final errors of the fit stored in it. As an example, here are the parameters used by the `EBV_model` class:

| Variable | Description |
|---|---|
| Tmax | Time of B maximum (days). |
| dm15 | The decline rate parameter, $\Delta m_{15}$ (mag.) |
| EBVhost | Host $E(B-V)$ reddening (mag.) |
| DM | Distance modulus (mag.) |

Table 4: Fit parameters of the `EBVmodel` instance.

The model can also have a variable number of parameters, as is the case in `max_model`. In this case, you can dynamically set up `self.parameters` in the `self.setup()` function (see below).

## 8.2 self.__init__(self, parent)

Next, you will override the `self.__init()` function. The only two arguments are `self` and `parent`. `self.__init__` must also call the `model.__init__()` function as part of the initialization. Other than that, you are free to setup whatever member variables you want. `self.__init__()` is called when the `model` instance is created, that is, when the `sn` object is created or when `sn.choose_model()` is called.

## 8.3 self.setup(self)

There are cases when you need to do some setting up after the user has called `sn.fit()`, but before the actual fitting occurs. For example, in the case of `max_model`, the number of parameters depends on how many filters the user fits (each filter has its own maximum). So in `self.setup()`, `self.parameters` is updated. This is not needed in `EBV_model`, because it has a fixed set of parameters regardless of how many filters are fit. However, it *does* do some initial checking to make sure that if `EBVhost` is a free parameter, the user has asked to fit at least two filters.

## 8.4 self.guess(self, param)

Before the fitting starts, any un-initialized variables (those whose value are `None`) need to be set to valid values, presumably close to the actual solution. You must override `self.guess()` to do this. For any parameter `param`, return an initial guess for this parameter. The `self.guess()` function is called after `self.setup()`, but just before the fitting starts.

## 8.5 self.__call__(self, band, t, **args)

This is the meat of the model. Given a filter `band` and time `t` (which is an array of times), return the model for the light-curve as an array of floats. Now, the question comes up: should the model return magnitudes or fluxes? The answer is: either. You can choose to return a model in magnitudes or fluxes, but you need to set the member variable `self.model_in_mags` to `true` if your model returns magnitudes, or `false` if it returns fluxes. *SNooPy always does the actual least-squares fitting in fluxes.*

You can define any number of optional arguments after `t` (replace `**args` with your arguments). You can then specify these optional arguments in the `sn.fit()` call and they will propagate through to the `__call__()`. For example, `EBV_model` has the optional `calibration` parameter.

### 8.6 self.get_max(self, bands, restframe=0, deredden=0)

The last member function to override is get_max(). Given a set of filters (`bands`), return the model's value at maximum light. If the optional argument `restframe=1`, then apply a k-correction to the value. If `deredden=1`, remove any reddening (galactic and host, if it is defined).

## 9 Template class

### 9.1 dm15temp.py

This class is basically a wrapper to Prieto's template generator and is used to generate templates for the `sn` class. It is available to scripts by importing the `dm15temp` module. The constructor doesn't need any arguments, so you simply make an instance as follows:

```
t = template()
```

The instance then has member variables for each filter defining a template: `t.B, t.V, t.R`, and `t.I`. There are also errors in these quantities: `t.eB, t.eV, t.eR`, and `t.eI`. The epoch is contained a variable `t.t` Each of these variables is a python array. Immediately after creating the instance, there will be no template defined. One has to make it with a specific value of $\Delta m_{15}$:

```
t.mktemplate(1.1)
```

This will run Prieto's code and insert the proper values into the member variables. This is all done in C, so is quite fast. The only other member function is `t.eval(band, times, z=0)`. This function is used to evaluate the template at specific times (useful for doing least-squares fitting to data). Simply specify which filter as a string (`band`) and an array of epochs to evaluate (`times`). The function returns two values: an array of interpolated values of the template and a mask array. This mask will be 1 for interpolation and 0 for extrapolation (where you should not use the data). You can also specify a redshift z so that the times are interpreted as observed epochs and will be converted to rest-frame epochs before evaluating.

### 9.2 CSPtemp.py

This is new module and constitutes a new method very similar to Prieto's technique. The idea is that one has a set of N well-sampled light-curves with pre-maximum data, so that $\Delta m_{15}$, $T_{max}$, and $m_{max}$ are all well determined. One then has a set of data points that define a surface in the 3D parameter space: $(t - T_{max}, \Delta m_{15}, m - m_{max})$. The problem is that this surface is sparsely and heterogeneously sampled. Prieto's method solves this in two steps: 1) construct spline representations of the N light-curves in the $t-$direction, then interpolate in the $\Delta m_{15}$-direction by way of averaging the splines with an adaptive weight function.

This new generator uses an algorithm developed by Barry Madore called GLoEs (Gaussian Local Estimation). In 1-D, one simply interpolates by way of a quadratic (or higher-order polynomial) through all the available data points. However, the weights of the points are determined using a Gaussian centered at the desired interpolation point and with a width sufficient to include the minimum number of points required for the polynomial. In 2D, one uses an elliptical Gaussian in the same way and fits a 2D polynomial to the data points. The advantage is that this is done in one step and data can be added without the need to re-train the fitter. The disadvantages are: 1) it's a slower process and 2) the resulting templates have more freedom to deviate from the idealized behaviour. For instance, if one asked for a template with $\Delta m_{15} = 1.15$, and actually measured $\Delta m_{15}$ directly, you would get a slightly different answer. As such, $\Delta m_{15}$ becomes a parameter, rather than a direct measurable.

Despite these very different approaches, the `CSPtemp.py` module behaves exactly the same as the older dm15temp.py module as far as the user is concerned. They can be used interchangeably. However, CSPtemp.py uses the CSP dataset, so offers a different set of filters.

### 9.3 ubertemp.py

In order to facilitate mixing-and-matching of different filters, a module had to be created that would allow the user to pick either the CSP or Prieto filters. This is `ubertemp`. The way you choose which filters to use is by name. Asking for a filter in the set `u,B,V,g,r,i,Y,J,H` will generate CSP templates. Asking for a filter in the set `Bs,Vs,Rs,Is` will generate Prieto templates (think of the 's' as 'standard' system, as opposed to the CSP templates which are in the CSP natural system). So, for example, you could compute a poor-man's S-correction between the CSP natural B-band and the standard B-band:

```
In[1]:  t = ubertemp.template()
In[2]:  t.mktemplate(1.4)
In[3]:  B_CSP = t.eval('B', arange(-10,60,1.0))
In[4]:  B_std = t.eval('Bs', arange(-10,60,1.0))
In[5]:  Scorr = B_CSP - B_std
```

As far as fitting goes, simply use the `restbands` member variable to choose which filter you wish to fit with. The model will then take care of generating the appropriate template and fit it to your observations.

## 10 Filters and Spectra

Two other useful classes are `filter` and `spectrum`. They are available for scripting by importing the `filters` module. The filter class inherits from the spectrum class, which is the more general object (a filter can be considered a spectrum of sorts).

### 10.1 spectrum object

You create a spectrum instance with two optional arguments:

```
spec = spectrum(name, file)
```

The name is just an identifying string. The file contains the spectrum as (lambda, flux) pairs, one per line. The member variables are given in the following table.

| name | type | description |
|---|---|---|
| `name` | string | Descriptive name for the spectrum |
| `file` | string | Filename of the spectral data |
| `wave` | float array | Array of wavelengths |
| `resp` | float array | Array of fluxes or responses (for filters) |
| flux | float array | Alias for 'resp' |
| comment | string | Any useful comments you want to ad |
| wavemax,wavemin | float | The minimum and maximum wavelengths defined by this spectrum |
| avewave | float | The average wavelength (useful for filters) |

Table 5: Member variables of the spectrum class.

## 10.2 Filter object

The filter object inherits from the spectrum object, so it has all its member variables. The filter is created in the same way the spectrum is, except it has one additional optional argument:

```
f = filter(name, file, zp)
```

The argument zp is the zeropoint of this filter. If you know it beforehand, specify it here. Otherwise, you can use the compute_zpt() member function described below. The zero-point is stored as the member variable `f.zp`. The filter class also adds a few extra member functions, which are described below:

- `compute_zpt(spectra, mag, zeropad=0)`: Given a single spectrum or list of spectrum instances (`spectra`) and a list of associated magnitudes (`mag`), compute the zero-points of this filter for these spectra, which are returned as an array. You could average the output to get a good handle on the actual zero-point. If the wavelength range of the filter is not completely inside the wavelength range of the spectrum, an exception is raised, unless `zeropad=1`, in which case the spectrum is assumed to be 0 outside the filter range.

- `response(wavespec, flux=None, z=0, zeropad=0, photons=1)`: Given an array of spectra, compute the response of the filter across the spectra: $\int F(\lambda)S(\lambda)\lambda/ch\,d\lambda$ (if `photons=1`), or $\int F(\lambda)S(\lambda)\,d\lambda$ if `photons=0`. You can either specify the spectra as arrays of spectrum instances and leave `flux=None`, or else specify an array of wavelength arrays and flux arrays. If the optional redshift, `z`, is given, the spectrum is red-shifted before the integration is done (actually, the filter is blue-shifted). zeropad has the same meaning as in compute_zpt().

- `synth_mag(wavespec, flux=None, z=0, zeropad=0, photons=1)`: Given an array of spectra, compute the synthetic magnitude through this filter. The zero-point must be defined in the instance (use `compute_zpt()` if needed). If a redshift is specified, the filter is blue-shifted by $1/(1+z)$ before computing the response (as in `response()`). Arguments are the same as `response()`.

## 10.3 Default spectra and filters

The `filters` module has two variables: `fset` and `spectra`. `fset` (filter set) is a dictionary-like object of pre-defined filters for your use. The filters have pre-defined zero-points, so can be used to compute synthetic magnitudes "out of the box" (see the accompanying document `zeropoints.pdf` for a discussion on how there are determined). The filters are organized by observatory/telescope/filter. You can list the observatories using fset.list_observatories(). You then use the observatory name as a member variable to get the observatory. Each observatory has a function list_telescopes(). The telescope name is used as a member variable to get the telescope, which has a list_filters() function. Each filter can also have a unique string ID that can be used directly. Here are some examples to give you an idea how it works.

```
print fset.list_observatories
print fset['B']
print fset.LCO.Swope.B
print fset.B
print fset.LCO.list_telescopes()
print fset.LCO.Swope.list_filters()
```

These filters are loaded at runtime and come from data files distributed with SNooPy. The are located in the source distribution in the folder `SNooPy/filters/filters`. In that folder are folders for each observatory. In each observatory foder, there are folders for each telescope/instrument, and in each telescope/instrument folder, there are files that contain the filter bandpasses and a file named `filters.dat`. Here is a portion of the folder structure:

```
|-- filters
|   |-- APO
|   |   -- SDSS
|   |       |-- filters.dat
|   |       |-- sdss_g.dat
|   |       |-- sdss_i.dat
|   |       |-- sdss_r.dat
|   |       |-- sdss_u.dat
|   |       |-- sdss_z.dat
|   |-- CFHT
|   |   -- Megacam
|   |       |-- filters.dat
|   |       |-- g_snls.dat
|   |       |-- i_snls.dat
|   |       |-- r_snls.dat
|   |       |-- z_snls.dat
```

The filters.dat file has a table of the filters, their names, and zero-points. Here is a sample of the SDSS filters.dat file:

```
u_s sdss_u.dat 12.4757864 sloan u at APO
g_s sdss_g.dat 14.2013159905 sloan g at APO
r_s sdss_r.dat 14.2156544329 sloan r at APO
```