A Guide to PHOEBE

Markus Rasmussen Mosbech

June 27, 2017

Abstract

PHOEBE[1] is a tool for caclulating physics of eclipsing binaries. It has a great number of options for fitting data and uses an intuitive Python front-end. In this guide, I aim to provide an introduction to the basics of PHOEBE with instructions regarding how to use its front-end. My experience with learning to use PHOEBE comes from reading the online tutorials at phoebe-project.org and this will likely be reflected in this guide.

For more advanced topics, I suggest the advanced tutorials on the PHOEBE website.

For a full list of supported and unsupported features, see http://phoebe-project.org/docs/2.0/.

[1] When I write PHOEBE, I refer to PHOEBE 2

Contents

Chapter 1

The Basics

1.1 Installation

PHOEBE requires Python 2.7x (not tested with Python 3.x) with the packages numpy (1.10+), scipy and astropy (1.0+). Matplotlib and sympy are also recommended. A compiler supporting C++11, such as g++5 is also required for the C++ included in PHOEBE.

The easiest way to install PHOEBE is from PIP with the command

```
pip install phoebe
```

To update an existing PHOEBE installation use

```
pip install –U phoebe
```

It is also possible to download the source code from GitHub and compile PHOEBE yourself. For more information on downloading and installing, see http://phoebe-project.org/docs/2.0/#download-and-installation.

If there are problems with importing PHOEBE in python, make sure your c++ compiler is actually up to date. If using a default version of Anaconda, you may have to either update Anaconda, if not up to date, or update specific components.

1.2 Logger

When writing scripts, especially when not working in a Jupyter notebook, it can be advantageous to have a logger. This logger determines what is printed in the terminal and what is saved to a logfile. Creating the logger might look like this

```
logger = phoebe.logger(clevel='WARNING',
        flevel='DEBUG', filename='tutorial.log')
```

Here everything with importance level WARNING or above is printed to the terminal, while everything DEBUG level or above is printed to tutorial.log. The information levels from most to least information are

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

1.3 Parameters

A python script or Jupyter notebook using PHOEBE will likely begin with importing phoebe and other necessary packages.

```
import phoebe
from phoebe import u
import numpy as np
import matplotlib.pyplot as plt
```

Be aware that the units built into PHOEBE are those defined in the IAU 2015 Resolution. These are in conflict with those defined in the astropy package, therefore, you should refrain from using units from astropy when using PHOEBE.

1.3.1 Parameter Types

PHOEBE uses a type of data called parameters. The parameter datatype holds a single value, which can be one of several types. A StringParameter can be initialized with the command

```
param = phoebe.parameters.StringParameter(
                qualifier='myparameter',
                description='mydescription',
                value='myvalue')
```

This StringParameter thus has a qualifier, a description and a value. The qualifier is a kind of abbreviated name of the parameter, the description describes the variable and the value is the value of this parameter, in this case a string.

The next type of parameters, ChoiceParameters are slightly more involved, though still very like StringParameters. A ChoiceParameter is essentially a StringParameter where the value has to be one of a specific set of possible values. It can be initialized like this:

```
param = phoebe.parameters.ChoiceParameter(
                qualifier='mychoiceparameter',
                description='mydescription',
                choices=['choice1', 'choice2'],
                value='choice1')
```

Here 'choices' is a list of all possible values for this parameter, in this case 'choice1' and 'choice2'.

FloatParameters are likely the most used PHOEBE parameter, they hold a float and a unit. This makes it possible to get the value of this parameter in any unit you want. It is initialized like this

```
param = phoebe.parameters.FloatParameter(
                qualifier='myfloatparameter',
                description='mydescription',
                default_unit=u.m,
                limits=[None,20],
                value=5)
```

This parameter thus has the value of 5 metres. Note that the PHOEBE units package has been used. The 'limits' attribute denotes the boundaries on the possible values. If the value is later changed, it cannot be larger than 20 metres, though there is no lower bound.

IntParameters are very much like FloatParameters, but the value must be an integer and has no units. It can be initialized like this

```
param = phoebe.parameters.IntParameter(
                qualifier='myintparameter',
                description='mydescription',
                limits=[0,None],
                value=1)
```

Having a value of 1, a lower bound of 0 and no upper bound.

Even simpler than IntParameters are BoolParameters, they only accept True or False. They generally work like Python booleans, an empty string will cast to

False while a non-empty string will cast to true. There is an exception however, both of the strings `'False'` and `'false'` will cast to False. It can be initialized like this:

```
param = phoebe.parameters.BoolParameter(
                qualifier='myboolparameter',
                description='mydescription',
                value=True)
```

leading to a BoolParameter with the value True.

The last parameter type is FloatArrayParameters, these are similar to FloatParameters, but instead of a float, the value is a numpy array of floats. Initialized like this

```
param = phoebe.parameters.FloatArrayParameter(
                qualifier='myfloatarrayparameters',
                description='mydescription',
                default_unit=u.m,
                value=np.array([0,1,2,3]))
```

1.3.2 Manipulating Parameters

There are various ways of working with parameters. One of the simplest operations is printing the parameter, this is done with the usual Python command **`print`**, followed by the parameter name. This will print various information stored in the parameter. The information stored in a parameter can be seperated into two types, attributes and tags (though both are attributes in a pythonic sense). The tags define the parameter and how it connects to other parameters. All parameter types have the same tags. Attributes however, are dependent on the parameter type and determines how to interpret the data. Both tags and attributes can be extracted from a parameter using either a dictionary key or an object attribute (the regular Python meaning of attribute). Thus, the result of both of these operations is the qualifier of the parameter

```
param.qualifier
param['qualifier']
```

A list of all tags can be accessed with

```
param.meta
```

and all attributes with

```
param.attributes
```

With param being the name of the parameter as a Python object for all these examples.

The value attribute has some additional functionalities, in part because this is the only attribute you're likely to want to change from its initial value. For a StringParameter, you can change the value with

```
param.set_value('newvalue')
```

and extract it with

```
param.get_value()
```

Some additional options are available for FloatParameters, such as getting the value in any unit.

```
param.get_value(unit=u.km)
```

This will yield the value in units of km. If no unit is specified, the default unit of the parameter will be used. A unit is also accepted when setting the value. If no unit is provided, the default unit will be assumed. The following sets the value using a specific unit, in this case km.

```
param.set_value(0.001*u.km)
```

These data can also be accessed as astropy quantities, a type of object containing both a value and a unit. This is done with either of

```
param.get_quantity()
param.get_quantity(unit=u.km)
```

The former uses the parameter's default unit, while the latter uses the provided unit, in this case km. The default unit can be changed with

```
param.set_default_unit(u.km)
```

here changing the default unit to km.

1.4 ParameterSets

A ParameterSet is a collection of parameters. It can be filtered by tags to return new ParameterSets. Here is an example of defining 3 parameters and then using them to create a ParameterSet

```
param1 = phoebe.parameters.FloatParameter(
                qualifier='param1',
                description='param1 description',
                default_unit=u.m,
                limits=[None,20],
                value=5,
                context='context1',
                kind='kind1')

param2 = phoebe.parameters.FloatParameter(
                qualifier='param2',
                description='param2 description',
                default_unit=u.deg,
                limits=[0,2*np.pi],
                value=0,
                context='context2',
                kind='kind2')

param3 = phoebe.parameters.FloatParameter(
                qualifier='param3',
                description='param3 description',
                default_unit=u.kg,
                limits=[0,2*np.pi],
                value=0,
                context='context1',
                kind='kind2')

ps = phoebe.parameters.ParameterSet(
                [param1, param2, param3])
```

Here ps is a ParameterSet consisting of the three parameters param1, param2, and param3. Printing the ParameterSet using the **print**-command yields a list of the three parameters and their values with their units displayed.

1.4.1 Twigs

The command

```
print ps
```

with the previously defined ps, yields

```
ParameterSet: 3 parameters
              param1@kind1@context1: 5.0 m
              param2@kind2@context2: 0.0 deg
              param3@kind2@context1: 0.0 kg
```

The string used to identify the parameters is called a twig. It is a combination of all the tags of the parameter, joined together with @s. It is also a way to access a parameter.

In order to access parameters in a ParameterSet using twigs, use

```
ps.get('param1@kind1')
```

This returns the ParameterObject with this unique combination of tags, in this case param1. When accessing a parameter using a twig, you only need to provide enough tags to get a unique match, thus both `'param1'` and `'kind1'` would have been enough to get param1, since these tags are both unique to param1. However, `'kind2'` would have to be used in conjunction with another tag to get a unique match.

As this returns the ParameterObject itself, any parameter methods or attributes can be used here. For instance

```
print ps.get('param1@kind1').description
```

yields the description of param1. It is also possible to get and set values of parameters in the set, using just the ParameterSet. The commands

```
ps.set_value('param1@kind1', 10)
ps.get_value('param1@kind1')
```

respectively set and get the value of param1.

1.4.2 Tags

Jut like parameters have tags, ParameterSets also have tags. The value of a ParameterSet tag is `None` if it is not shared among all parameters in the set. A list of all the tags can be printed using

```
print ps.meta.keys()
```

yielding

```
['time', 'qualifier', 'history', 'feature',
 'component', 'dataset', 'constraint', 'compute',
 'model', 'fitting', 'feedback','plugin', 'kind',
 'context']
```

These tags often act as labels for easier reference, however, some are fixed:

- qualifier: literally the name of the parameter.
- kind: tells what kind a parameter is (ie whether a component is a star or an orbit).
- context: tells what context this parameter belongs to
- twig: a shortcut to the parameter in a single string.
- uniquetwig: the minimal twig needed to reach this parameter.
- uniqueid: an internal representation used to reach this parameter

The contexts are:

- setting
- history
- system
- component
- feature
- dataset
- constraint
- compute
- model
- fitting [not yet supported]
- feedback [not yet supported]
- plugin [not yet supported]

The online PHOEBE tutorial has an excellent explanation of the difference between kinds and contexts, I bring it here in full:

> One way to distinguish between context and kind is with the following question and answer:
> "What kind of [context] is this? It's a [kind] tagged [context]=[tag-with-same-name-as-context]."
>
> In different cases, this will then become:
> "What kind of component is this? It's a star tagged component=starA."
> (context='component', kind='star', component='starA')
>
> "What kind of feature is this? It's a spot tagged feature=spot01."

(context='feature', kind='spot', feature='spot01')

"What kind of dataset is this? It's a LC (light curve) tagged dataset=lc01."
(context='dataset', kind='LC', dataset='lc01')

"What kind of compute (options) are these? They're phoebe (compute options) tagged compute=preview."
(context='compute', kind='phoebe', compute='preview')

Tags of ParameterSets can only be accessed as object attributes, not as dictionary entries, unlike the case for parameters. For instance, to get the context of the ParameterSet ps, the only option is

```
print ps.context
```

This will return None, because not all parameters in the set have the same context, as mentioned earlier. However, calling the attribute in plural

```
print ps.contexts
```

will yield a list of all the contexts included in the ParameterSet, in this case

```
['context2', 'context1']
```

1.4.3 Filtering

ParameterSets can be filtered using tags, for instance

```
print ps.filter(context='context1')
```

will return

```
ParameterSet: 2 parameters
               param1@kind1@context1: 10.0 m
               param3@kind2@context1: 0.0 kg
```

That is, the ParameterSet containing all parameters with `context='context1'` is returned and printed. The filter command can also filter using multiple tags, seperated by commas, such as

```
print ps.filter(context='context1', kind='kind1')
```

This will return a ParameterSet containting only param1, since this is the only parameter matching the filter. Note however, that the filter will always return a ParameterSet, even if it contains only one parameter. To access the parameter, `get` must be used. Either instead of the filter, or appended to it. Thus the two calls

```
ps.filter(context='context1', kind='kind1').get()
ps.get(context='context1', kind='kind1')
```

will both return the param1 parameter.

This can also be achieved using twigs. The command

```
        ps['context1']
```

will return a ParameterSet containing the parameters with `context='context1'`. These can also be chained, returning ParameterSets containing parameters matching multiple filters, this is done with either of

```
ps['context1']['kind1']
ps['context1@kind1']
```

However, using twigs is different from filters in one way. If only one parameter matches the conditions, the parameter is returned rather than a ParameterSet containing it.

1.5 Bundles

Bundles are similar to ParameterSets, but contain additional functionalities, such as methods to compute models and add components. An empty Bundle can be initiated with

```
b = phoebe.Bundle()
```

Printing a Bundle yields a list of contexts and which parameters are part of these contexts. Printing this empty Bundle yields

```
print b
```

```
SYSTEM:
distance
hierarchy
t0
vgamma
epoch
ra
dec

COMPONENT:

DATASET:

CONSTRAINT:

COMPUTE:

MODEL:

FITTING:

FEEDBACK:

PLUGIN:
```

The Bundle can be filtered using both the filter command and by using twigs as dictionary entries, just as with ParameterSets. It is possible to manipulate parameters in a bundle, either by accessing them by their twig, then manipulating them, or by telling the bundle to manipulate it, using the twig. Thus, these two calls are equivalent

```
b['distance'].set_value(100)
b.set_value('distance',100)
```

Bundles also make use of the parameter attribute visible_if. This makes the visibility of parameters conditional, depending on other parameters. Initializing a bundle from two parameters

```
param1 = phoebe.parameters.ChoiceParameter(
                qualifier='what_is_this',
                choices=['matter', 'aether'],
                value='matter',
                context='context1')
```

```
param2 = phoebe.parameters.FloatParameter(
                qualifier='mass',
                default_unit=u.kg,
                value=5,
                visible_if='what_is_this:matter',
                context='context1')

b = phoebe.Bundle([param1, param2])
```

If printing Bundle b now, both parameters will show up, but this is because the value of param1, `'what_is_this'`, is `'matter'`. If it had been `'aether'`, `'mass'` would not show up. This is for hiding irrelevant parameters, like here. If 'this' is matter, then it will have mass, but if it is aether, it is massless and thus there is no need for a mass parameter.

It is possible to save a Bundle to a file in JSON format, this is done with

```
b.save('filename.phoebe')
```

and saved bundles can be opened with

```
b2 = phoebe.Bundle.open('filename.phoebe')
```

1.5.1 Interpolation

If a FloatArrayParameter is part of a Bundle, it is possible to perform simple linear interpolation in it with respect to another FloatArrayParameter in the same bundle. Initializing a Bundle from two parameters

```
xparam = phoebe.parameters.FloatArrayParameter(
                qualifier='xs',
                default_unit=u.d,
                value=np.linspace(0,1,10),
                context='context1')

yparam = phoebe.parameters.FloatArrayParameter(
                qualifier='ys',
                default_unit=u.m,
                value=np.linspace(0,1,10)**2,
                context='context1')

b = phoebe.Bundle([xparam, yparam])
```

it will be possible to interpolate, such that

```
b['ys'].interp_value(xs=0.2)
```

yields the value of `'ys'` at `xs=0.2`, in this case ~ 0.04 (the remaining digits are omitted for brevity). Not however, that the interpolation yields values without units, not astropy quantities. The default units are always used.

Chapter 2

Building a System

A system can be built by initializing an empty Bundle as previously shown and then adding parameters to it, or by using a default bundle. A good way to start when building a system is by initializing the default binary Bundle, this can be done with either of these calls:

```
b = phoebe.Bundle.default_binary()
b = phoebe.default_binary()
```

This initializes a Bundle with containing various parameters describing two stars and their orbit. It is possible to build a contact binary system by adding `contact_binary=True` in the parenthesis. Using a default system Bundle might be a bit overwhelming however, as the Bundle contains 106 different parameters across five contexts. It might be useful to play around with, however, to see how a fully formed Bundle for a system looks. It is also a good way to identify useful parameters.

2.1 Adding Components

To build a system from an empty Bundle, first initialize an the Bundle. When the Bundle is initialized, it is time to add components to the system. Most components have to be added with the `add_component` method. Here I add two stars to the bundle with different syntax

```
b.add_component(phoebe.component.star,
                component='primary')
b.add_component('star', component='secondary')
```

The component tags `'primary'` and `'secondary` are just labels. They do not hold any specific meaning, they might as well have been `'Star1'` and `'Star2'` or `'Trevor'` and `'George'`.

There are also shortcuts for adding stars and orbits. Using these, it is also possible to provide values for any qualifiers of the created object. To add a star with component tag `'ThirdStar'` and an effective temperature of 5000 K, use

```
b.add_star('ThirdStar', teff=5000)
```

An orbit can be added with

```
b.add_orbit('binary')
```

Again `'binary'` is just a label, it has no influence on the added orbit other than the identifier.

In conclusion the `add_component` method can take several arguments. First, an argument telling what kind of component is added and thus which parameters to add, second the tag to identify the component, and third values for created parameters, such as `teff` in the example.

2.2 Hierarchy

In addition to all the parameters added with the components, a system needs a hierarchy, specifying the setup of the system.

To place the stars into the binary orbit, use either of

```
b.set_hierarchy(phoebe.hierarchy.binaryorbit,
      b['binary'], b['primary'], b['secondary'])
b.set_hierarchy(phoebe.hierarchy.binaryorbit(
      b['binary'], b['primary'], b['secondary']))
b.set_hierarchy(
      'orbit:binary(star:primary, star:secondary)')
```

The latter is the same syntax as is used when printing the hierarchy, done using

```
b.get_hierarchy()
```

For a system with more than two stars, an orbit can be used as a component of a different orbit. After adding an additional outer orbit, the hierarchy of the tertiary system can then be set with

```
b.set_hierarchy('orbit:outer(orbit:binary(star:primary,
star:secondary), star:ThirdStar)')
```

This is then an outer orbit whose two components are the the binary system and a third system. This is then effectively a triple system. Be aware that triple systems are not yet fully supported, so make sure that your results make sense in the context.

There are various commands for getting different parts of the hierarchy, most are fairly self-explanatory.

```
b.hierarchy.get_stars()
b.hierarchy.get_orbits()
```

These will return a list of either all stars or all orbits in the hierarchy.

There are also methods for getting components with specific relations to other components:

```
b.hierarchy.get_parent_of(component)
b.hierarchy.get_children_of(component)
b.hierarchy.get_child_of(component, index)
```

The first returns the parent compoent of the input component. If `'binary'` from our triple system is the input, it would yield `'outer'`. The second returns all children of the input component. If `'binary'` is the input, it yields `['primary','secondary']`, while `'outer'` would yield `['binary','ThirdStar']`. The third returns a specific child, either the primary component or the secondary, depending on the index (0 for primary, 1 for secondary).

Finally there are

```
b.hierarchy.get_sibling_of(component)
b.hierarchy.get_primary_or_secondary(component)
```

the first of these returns the sibling of the input component, if any (otherwise it returns None). The second returns whether the input component is the primary or secondary component in its orbit.

2.3 Constraints

Constraints have their own context in a Bundle. They are often created when components are added or the hierarchy is set. When initializing a default binary system, it will be initialized with a total of 20 constraints. To see a constraint, it can be printed like any other parameter. For instance the `'mass@primary@constraint'`-constraint calculates the mass of the primary component from the orbit parameters.

If printing the primary mass `'mass@primary@component'`, it can also be seen exactly which parameters are used to constrain it.

```
print b['mass@primary@component']
```

```
Parameter: mass@primary@component
        Qualifier: mass
        Description: Mass
        Value: 0.998813135806 solMass
        Constrained by: sma@binary@component,
                        period@binary@component,
                        q@binary@component
        Constrains: None
        Related to: sma@binary@component,
                    period@binary@component,
                    q@binary@componen
```

It can also be seen that it is not used to constrain any other parameters. Any other components constrained by this one are listed in Constrains.

2.3.1 Built-in Constraints

There are a number of built-in constrains, automatically applied when you build a system. These are

- **asini**
  Computes asini (the projected semi-major axis along the line of sight). Uses sma and incl (semi-major axis and inclination) Can be inverted to constrain either sma or incl.

- **esinw, ecosw**
  Calculates the projected eccentricity from ecc and per0 (the ecceptricity and argument of periastron).

- **t0**
  Handles conversion between different t0 conventions, providing reference time at periastron passage (t0_perpass) and superior conjunction (t0_supconj). Default is periastron passage, can be inverted to solve for superior conjunction.

- **freq**
  Handles conversion from period to frequency. Used for both rotation and orbits. Can be inverted to calculate period from frequency.

- **mass**
  Solves for the component mass by using Kepler's third law. Constrained by sma, period, and q (the mass ratio). Can be inverted to solve for sma or period, but not q.

- **component sma**
  Computes the semi-major axis of a component about the center of mass of its parent orbit. Be aware that this is not the same as the sma of the parent orbit. It is constrained by the sma of the parent orbit and q. It can be inverted to solve the parent sma instead.

- **pot**
  Handles solving for the roche surface equipotential from the polar radius, it can be inverted to solve for the polar radius rpole instead.

- **rotation period**
  Computes the rotation period of a star from its synchronicity parameter (syncpar). It can be inverted to solve for eiter orbital period or syncpar.

- **inclination**
  This constraint handles the requirement that the inclination of a star's rotation is the same as the system inclination. That is, makes sure the system is an aligned binary. Misaligned binaries are not supported at this time. It can be inverted to solve for either inclination.

2.3.2 Reparameterizing Constraints

As mentioned, various constraints can be inverted so that the parameters that are by default calculated from other parameters, can instead be used to calculate the parameters they would be derived from by default. This is an experimental feature, so it is advised to be careful and make doubly sure that results and parameters make sense.

To determine what a parameter is constrained by, the easiest way is probably either to print the parameter using its twig, or going specifically for its constraints. This is done (here for the mass of the primary star) with

```
b['mass@primary@component'].constrained_by
```

This will return the parameters used to calculate the mass of the primary star of the system. One of these is the period. In order to calculate the period from the mass instead, use the command

```
b.flip_constraint('mass@primary', 'period')
```

The mass of the primary component can now be set, and the period will change accordingly. Additionally the mass of the secondary component will change because they are related by the q parameter.

Constraints are named for the component that they constrain, therefore a different twig is required to flip back. The required twig in this case would be

```
b.flip_constraint('period@binary', 'mass')
```

Chapter 3

Datasets

Datasets are a crucial part of PHOEBE. They are required to work with observational data and computing synthetic models. To add a dataset, it is necessary to use the function in `phoebe.parameters.dataset` for that specific type of data. The available methods for datasets include:

- orb (orbit/positional date)
- mesh (discretized mesh of stars)
- lc (light curve)
- rv (radial velocity)

3.1 Adding Datasets

Adding an entirely artificial dataset is very simple, as an example, for a Bundle b with a default binary, adding specific timesteps to in an orbit can be done with

```
b.add_dataset(phoebe.dataset.orb,
                times=np.linspace(0,10,20),
                dataset='orb01',
                component=['primary', 'secondary'])
```

This makes it possible to track positions and radial velocities at each of these timesteps. `phoebe.dataset.orb` can also be replaced with `'orb'`, and the same applies to any other dataset type. The name as a string can be used instead of the `phoebe.dataset`-notation. To add the same timesteps for radial velocities, the following can be used

```
b.add_dataset('rv', times=np.linspace(0,10,20),
                dataset='rv01')
```

When not providing specific components, which components to use will be assumed based on which type of dataset is added. This will also lead to a `_default`-component being added. This component takes care of adding these times for rvs for any components later added to the system, where this is applicable. The `_default`-component is empty for the orb dataset, leading any new components to be added with empty orb datasets.

Light-curves and meshes always attach on a system-level, whereas rvs and orbs attach to individual stars.

Adding observed data is just as easy, to add a light-curve for the system, you can use

```
b.add_dataset('lc', times=[0,1], fluxes=[1,0.5],
                dataset='lc01')
```

The data will be added for all components where the time is applied. This makes good sense for light-curves but may not be the desired behaviour for datasets like radial velocities. A single-lined rv-dataset for one component is pretty straightforward to add

```
b.add_dataset('rv', times=[0,1], rvs=[-3,3],
                 dataset='rv02', component='primary')
```

This will result in these radial velocities being added to the primary component and none to the secondary. If no component was provided, the same radial velocities would be applied to both the primary and the secondary star. This is most likely not desired. To explicitly add different radial velocities to the two component stars, the following notation is used

```
b.add_dataset('rv', times=[0,1],
        rvs={'primary': [-3,3], 'secondary': [4,-4]},
         dataset='rv04')
```

A different way of doing this is by not passing values while adding the dataset, but rather using `set_value` afterwards for the individual components.

Passband options work similarly to dataset columns. Sending it a single value will apply it to each component the time array is attached to, based on the list of components sent or dataset method defaults. It is slightly different for light-curves however, as they are always system-level, but the passband options exist for individual stars. The value will therefore be passed to each star if a component is not explicitly provided. An example here is providing a limb darkening function and coefficients

```
b.add_dataset('lc', times=[0,1],
        ld_func='logarithmic', ld_coeffs=[0,0],
        dataset='lc02')
```

This will apply the function and coefficients to all stars. To pass different values to different components, provide them in a dictionary, as was also demonstrated for rvs

```
b.add_dataset('lc', times=[0,1], ld_func='logarithmic',
        ld_coeffs={'primary': [0,0],
        'secondary': [0.25, 0.25]},
        dataset='lc03')
```

Note that the `_default`-values were not overridden, so any stars added later will use the PHOEBE default values. To change this, you must also provide a value for `_default`.

To add data from a file is done in the same way. The data is loaded and saved in arrays outside of PHOEBE, such as with numpy. These arrays are then passed to PHOEBE, as in this example

```
times, fluxes, sigmas = np.loadtxt('test.lc.in',
        unpack=True)

b.add_dataset(phoebe.dataset.lc, times=times,
        fluxes=fluxes, sigmas=sigmas, dataset='lc04')
```

3.2 Removing Datasets

A list of datasets currently in a bundle can be accessed with `b.datasets`. The simplest way to remove a single dataset is by using its tag. To remove the dataset `'lc01'`, use

```
b.remove_dataset('lc01')
```

There are also ways to remove several datasets at once, the `remove_dataset` method accepts any tags that can be sent to filter. For instance, to remove all radial velocity datasets, use

```
b.remove_dataset(kind='rv')
```

Chapter 4

Computation

Once datasets are added to a Bundle, it is possible to compute a synthetic model for these datasets. The computation uses the PHOEBE backend, for advanced users it is also possible to use different backends.

4.1 Compute Options

Any default bundle has a set of default computation options, telling how to run the backend. Usually, it is good enough to just edit these options, but it is also possible to add new sets of computation options. This could be useful if you want to try different computation options for the same dataset. To see all computation options currently stored in a Bundle, just use one of the options for showing parameters. Many of these will be ChoiceParameters, allowing to choose from a specific set of options for this computations. For instance the compute parameter to handle irradiation effects has three possible options: `'wilson'`, `'horvat'` and `'none'`. The default is `'wilson'`. To change this, just use a `set.value` command, setting the value to one of the other options.

Adding a set of compute options to a Bundle is similar to adding anything else, to add add a set of compute options with tag `'detailed'` and irradiation method `'horvat'`, use either of

```
b.add_compute('phoebe', compute='detailed',
                irrad_method='horvat')
b.add_compute(phoebe.compute.phoebe,
                compute='detailed',
                irrad_method='horvat')
```

All compute options not explicitly set when adding the set will have use their default values. To see a list of all compute options and their values in a set, use either of

```
b['detailed@compute']
b.get_compute('detailed')
```

4.2 Choosing Datasets

By default, all datasets in the bundle are set to be used in all computation, but it is possible to disable datasets for specific sets of compute options. This is done with a BoolParameter called enabled, set to True or False for each set of compute options (True by defailt).

To see which compute options the light-curve dataset `'lc01'` is enabled for, use the twig

```
print b['enabled@lc01']
```

This will yield a ParameterSet containing the `'enabled'`-parameter for the `'lc01'`-dataset for each set of compute options, and its value. It can be set to True of False the same way as changing the value of any other parameter, for instance, setting it to false for the `'detailed'` set

```
b['enabled@lc01@detailed'] = False
```

or for all set of compute options

```
b['enabled@lc01'] = False
```

It can be reenabled by setting to True.

4.3 Running Compute

Synthetic models are computed using the `run_compute`. It takes arguments for the compute tag and the model tag of the resulting model. If there are 0 or 1 compute options in the Bundle, no compute tag is necessary. If the bundle has one compute option, this will be used, if it has none, the phoebe default compute option will be added and used. If no tag is provided for the synthetic model, it will be given the tag `'latest'`. Any new `run_compute` will overwrite old models if they have the same tag, therefore provide a tag if you want to store your model. To run compute, use

```
b.run_compute(compute='detailed', model='model1')
```

This will use the compute options `'detailed'` and give the computed model the tag `'model1'`. It will use all datasets in Bundle b enabled for `'detailed'`. To get a list of all computed models in a Bundle, use

```
b.models
```

It is also possible to run multiple computations with one `run_compute`, however, when doing this each dataset must only be enabled for one of the computes run. Running multiple computes can be done with

```
b.run_compute(compute=['detailed', 'preview'],
                  model='multiplecompute')
```

This creates one model with the results of both computes.

The synthetic models can be accessed through their twigs or through `get_value`.

Chapter 5

Plotting

Of course it is possible to access the arrays stored in Bundles and plot them manually, but PHOEBE also offers a number of built-in plotting options.

After having initialized a bundle, added data and computed a synthetic model, a default plot can be made with the call

```
axs, artists = b.plot()
```

This plots and returns a set of python axes and artists, in case you want to edit the plots further before saving them. In order to not plot everything, use tags to filter what to plot, such as

```
axs, artists = b['orb@run_with_incl_80'].plot()
```

This plots the orbit of the `'run_with_incl_80'` model.

It is also possible to highlight specific times in the plot. To make the same plot, but while highlighting time=1.0, use

```
axs, artists = b['orb@run_with_incl_80'].plot(time=1.0)
```

It is also possible to pass standard matplotlib colors, markers and marker-sizes. For instance, to make the markers large green squares, you can use `highlight_marker='s', highlight_color='g', highlight_ms=20`. If the tags or twig provided return multiple datasets with the same type of data, they will be plotted in the same plot. If there are different data types eg. positional data and a light-curve, sub-plots will be made for each data type.

When plotting, PHOEBE automatically uses the default arrays for each axis, in order to use different arrays, set the x- and y-axes to a specific array when calling plot, like this

```
axs, artists =b['orb01@primary@run_with_incl_80'].plot(
                x='times', y='vxs')
```

To plot in phases rather than time, use `x='phases'`, this will use the phase of the top level ephemeris. To use a specific level, use a colon to tell which, such as `x='phases:binary'`. This will use the ephemeris of the `'binary'`-component.

It is also possible to get phase data from Bundles. To get the ephemeris, use

```
b.get_ephemeris()
```

To calculate the phase at a time or vice-versa use

```
print b.to_phase(arg)
print b.to_time(arg)
```

Just inpute a time or phase as arg, depending on which you use. To get an entire dataset in phase rather than time, just use

```
b.to_phase(b['times@primary@orb01'])
```

This yields all of the timesteps in the `'times@primary@orb01'`-dataset, converted to phase.