PIPS

Release 0.3.0-alpha

Y. Murakami, A. Hoffman, J.Sunseri, A. Savel

Mar 04, 2023

GETTING STARTED

PIPS is a Python pipeline primarily designed to analyze the photometric data of astronomical objects whose brightness changes periodically. Our pipeline can be imported quickly and is designed to be user friendly. PIPS was originally developed to determine the periods of RR Lyrae variable stars and offers many features designed for variable star analysis. We have expanded PIPS into a suite that can obtain period values for almost any type of photometry with both speed and accuracy. PIPS can determine periods through several different methods, analyze the morphology of lightcurves via fourier analysis, and determine stellar properties based on preexisting stellar models. Currently our team is also exploring the possibility of using this pipeline to detect periods of exoplanets as well.

To determine the period value from provided data, PIPS runs a periodogram by evaluating the goodness of fit (chi-squared test) at a range of period values fitted to the data. PIPS will continue to refine the period estimation by re-running a periodogram test around a successively smaller range of values until a confidence threshold is reached. The goodness of fit is evaluated against a model and the period-folded data. PIPS offers choices of fitting models, with both Fourier and Gaussian Mixture currently implemented. PIPS can easily accept custom model functions to evaluate custom data while PIPS' preserving structure and ease of use. The choice of model is important for accuracy, and dependent on the type of data being analyzed. For instance, the relatively gradual changes of variable stars are well suited to Fourier models, which the sharp troughs of exoplanet light curves are much better fit by the Gaussian Mixture model.

Unlike many other period-determination methods, PIPS calculates the uncertainty of both the period and model parameters. Currently, this uncertainty is calculated using the covariant matrix of the linear regression. However, in the future the PIPS team intends to implement an optional MCMC based method which will offer further accuracy. While period estimation is the primary function of PIPS, other tools are available for specific uses. These include stellar parameter estimation, lightcurve-based classifications, and visualization helpers for data analysis. The combination of a robust data analysis structure, and the flexibility to work with external tools and models, ensures that PIPS is an excellent solution for a wide variety of periodic data analysis.

This notebook is available at https://github.com/SterlingYM/PIPS/tree/master/docs/installation.ipynb

CHAPTER

ONE

INSTALLATION

PIPS can be installed from PyPI or GitHub. It is recommended to use the latest version on PyPI, but users are encouraged to participate in development on GitHub or report issues and suggestions.

1.1 Installing with `pip`

[ ]:
```
# installing from PyPI -- for anyone!
!pip install astroPIPS
```

1.2 Installing from source

[ ]:
```
# installing from GitHub -- for advanced users
!git clone https://github.com/SterlingYM/astroPIPS
!cd astroPIPS
!python -m setup.py install
```

1.3 Check installed version

Once successfully installed, users should be able to run the following lines in Python:

[1]:
```
import PIPS
PIPS.__version__
```

[1]:
```
'0.3.0-alpha.4'
```

This notebook is available at https://github.com/SterlingYM/PIPS/tree/master/docs/tutorial.ipynb

CHAPTER

TWO

TUTORIAL

PIPS contains various tools for time-series analysis in astronomy, with primary focus on detecting the period of variability. PIPS is objectively programmed, so that the analysis can be performed in a straightforward way.

In this introductory tutorial, you will learn the quickest methods to do the following operations:

- Installing PIPS
- Initializing photometric data object — `PIPS.photdata`
- Generating periodogram — `photdata.periodogram()`: basic & advanced
- Detecting main period — `photdata.get_period()`: basic & advanced
- Quick visualization — `photdata.get_bestfit_curve()`: basic
- Multi-period analysis — `photdata.amplitude_spectrum()`: basic

2.1 Importing PIPS

PIPS is currently distributed on PyPI and GitHub under the name of `astroPIPS`. However, the package name itself is still under `PIPS`, and hence the import statements becomes as shown below:

[1]:
```
import PIPS
from PIPS.resources.sample_RRL import samples
import matplotlib.pyplot as plt
import numpy as np
import warnings
warnings.filterwarnings(action='ignore')
```

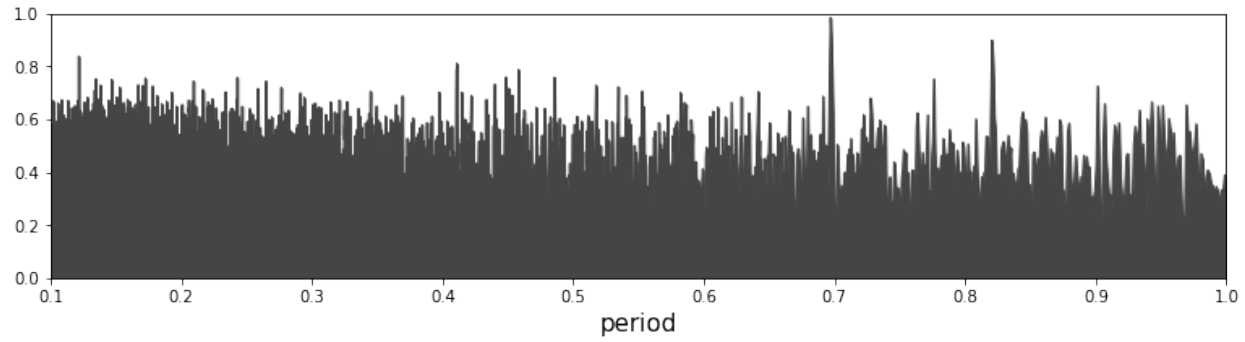
[2]:
```
PIPS.about()
```

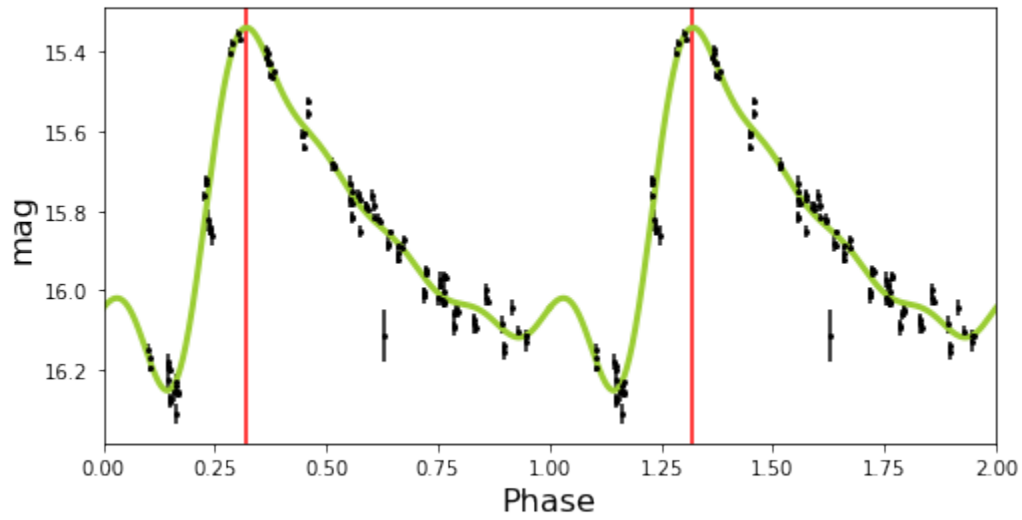
```
-------------------------
-    Welcome to PIPS!    -
-------------------------
Version: 0.3.0-beta.1
Authors: Y. Murakami, A. Savel, J. Sunseri, A. Hoffman, Ivan Altunin, Nachiket Girish
-------------------------
Download the latest version from: https://pypi.org/project/astroPIPS
Report issues to: https://github.com/SterlingYM/astroPIPS
Read the documentations at: https://PIPS.readthedocs.io
-------------------------
```

2.2 Before you start – sneak peek at PIPS in 5 lines

Photometry data to phase-folded light curve – This is what you can do with PIPS in 5 lines of code!

[13]:
```
# data prep & initialization
data = samples[0]
star = PIPS.photdata(data)

# generate periodogram
star.periodogram(p_max=1.0).plot()

# automatic period detection
star.get_period(N0=20)

# plot phase-folded data, best-fit curve, and time of maxima
star.plot_lc(plot_bestfit=True,plot_epoch=True)
```





PIPS is designed so that it can be as simple as this for basic analyses, but at the same time PIPS provides a powerful platform for more high-level analysis for professional astronomers. In the tutorial below, we go over the basic steps to perform some of the most frequent operations and analyses.

2.3 Data preparation

PIPS takes in an array of 3xN data (samples are available on github) – time, magnitude (flux), and error on magnitude contained in a single python list or numpy array. For convenience, photometry data file from LOSSPhotPipeline can be directly imported using a helper function `data_readin_LPP`.

[14]:
```
data = samples[0]
x,y,yerr = data
print('data shape:\t',np.array(data).shape)
print('x shape:\t',x.shape)
print('y shape:\t',y.shape)
print('y-error shape:\t',yerr.shape)
```

```
data shape:      (3, 103)
x shape:         (103,)
y shape:         (103,)
y-error shape:   (103,)
```

2.4 Create `photdata` object

Most of the functions in `astroPIPS` is implemented as methods in `photdata` object. Once the `photdata` object is initialized, various operations, such as period detection and data manipulation, can be done directly to the object.

[15]:
```
star = PIPS.photdata(data)
```

This object initially contains raw data, and as the user performs analyses using various functions, more information, such as cleaned data, period, or amplitude, will be stored.

The list of variables in the object can be printed with the following code:

[16]:
```
print('Initially defined variables: ')
# for att in dir(star): print('- ',att) )
[print('- '+att) for att in dir(star) if not callable(getattr(star, att)) and not att.
↪startswith('__')];

print('\nAvailable functions: ')
[print('- '+att+'()') for att in dir(star) if callable(getattr(star, att)) and not att.
↪startswith('__')];
```

```
Initially defined variables:
- amplitude
- amplitude_err
- band
- data
- epoch
- epoch_offset
- label
- meanmag
- multiprocessing
- period
- period_err
- shape
```

(continues on next page)

(continued from previous page)

```
- x
- y
- yerr

Available functions:
- _get_period()
- _get_period_likelihood()
- amplitude_spectrum()
- check_model()
- classify()
- copy()
- cut()
- get_SDE()
- get_SR()
- get_bestfit_amplitude()
- get_bestfit_curve()
- get_chi2()
- get_epoch_offset()
- get_meanmag()
- get_period()
- get_period_multi()
- open_widget()
- periodogram()
- plot_lc()
- prepare_data()
- reset_cuts()
- summary()
```

It is always a good idea to keep track of the name and photometric band of the object. For instance, the name of data file can be used as a label:

[17]:
```
star.label = '000.dat'
star.band  = 'V'
```

2.5 Generating periodogram

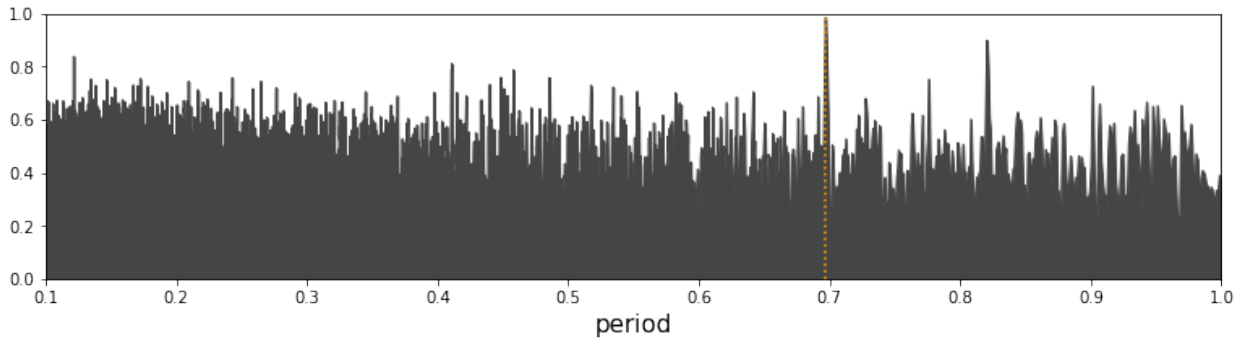
`periodogram()` function provides the most basic yet most valuable information on variability analysis. This is an extended application of Fourier transform in the period space (1/frequency). This function requires arguments `p_min` and `p_max`, which limits the range of periodogram (hence period search). The unit has to be the same (often *days* in astronomy) as the x-axis in the input data.

2.5.1 basic method

Most simply, users can call `star.periodogram(p_min,p_max).plot()` to generate periodogram.

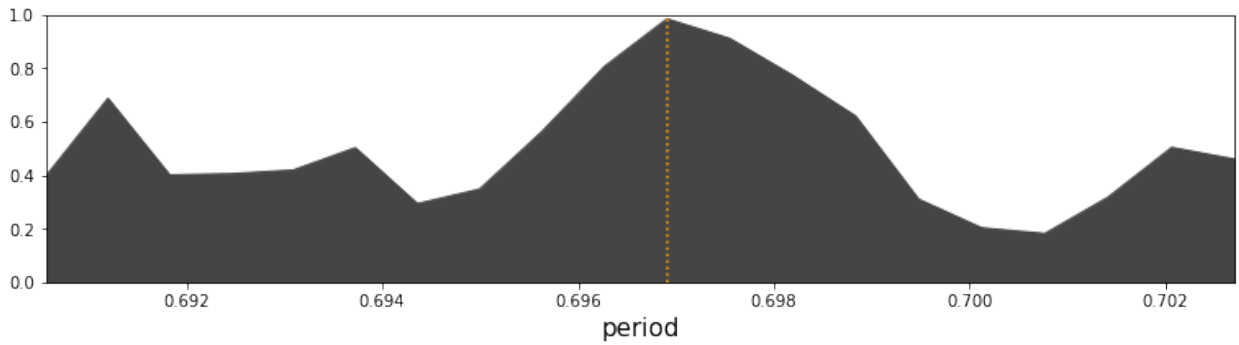
A few things to note: - This function shouldn't take more than a few seconds to run. If no result is returned, it may be because your platform does not support `multiprocessing` operation, in which case we recommend adding another argument `multiprocessing=False` to `periodogram()` function call. - If no `p_min` or `p_max` is given, periodogram is generated between 0.1 and 4 (days if your data is in days).

[22]:
```
# periodogram: searching the period between 0.1-day and 1-day
star.periodogram(p_min=0.1,p_max=1.0).plot(show_peak=True)
```

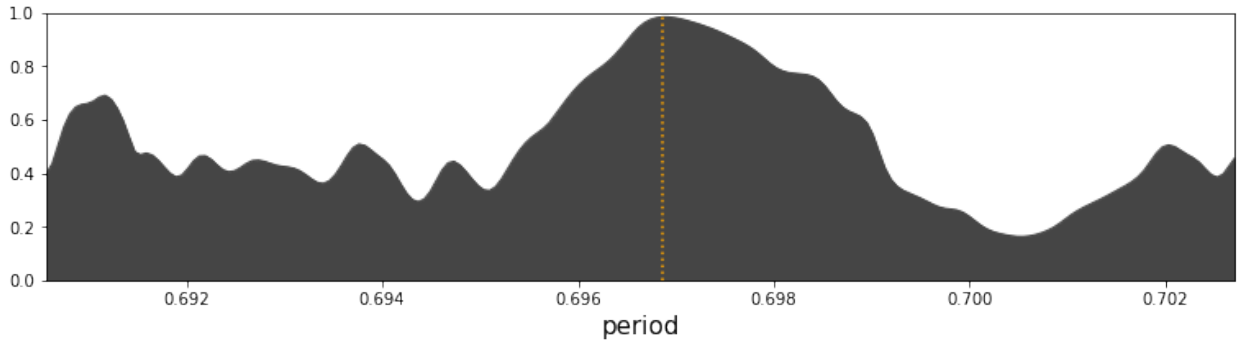


Zooming in to the peak and re-sampling can also be done in a single line:

[23]:
```
star.periodogram(p_min=0.1,p_max=1.0).zoom().plot(show_peak=True)
```



[24]:
```
star.periodogram(p_min=0.1,p_max=1.0).zoom().refine().plot(show_peak=True)
```

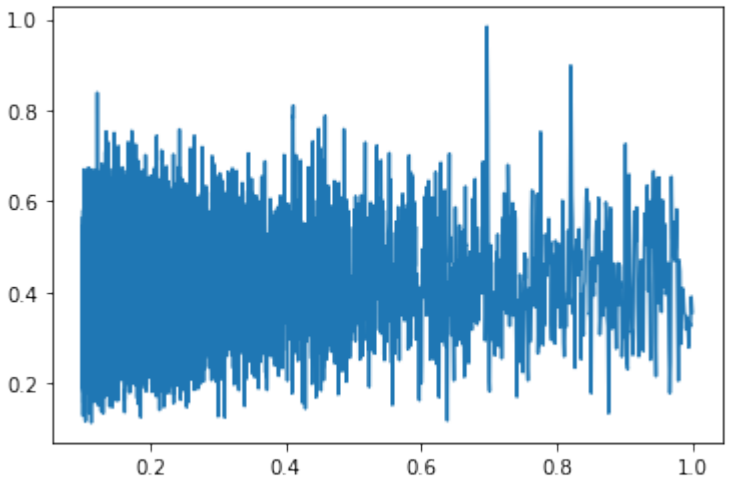


2.5.2 Obtain periodogram as arrays

The `periodogram()` function generates an iterable object. Users can easily obtain the values of periodograms and analyze or plot them manually.

[29]:
```
periods, power = star.periodogram(p_min=0.1,p_max=1.0)
plt.plot(periods,power);
print(periods.shape,power.shape)
```

```
(6830,) (6830,)
```



2.5.3 More advanced method

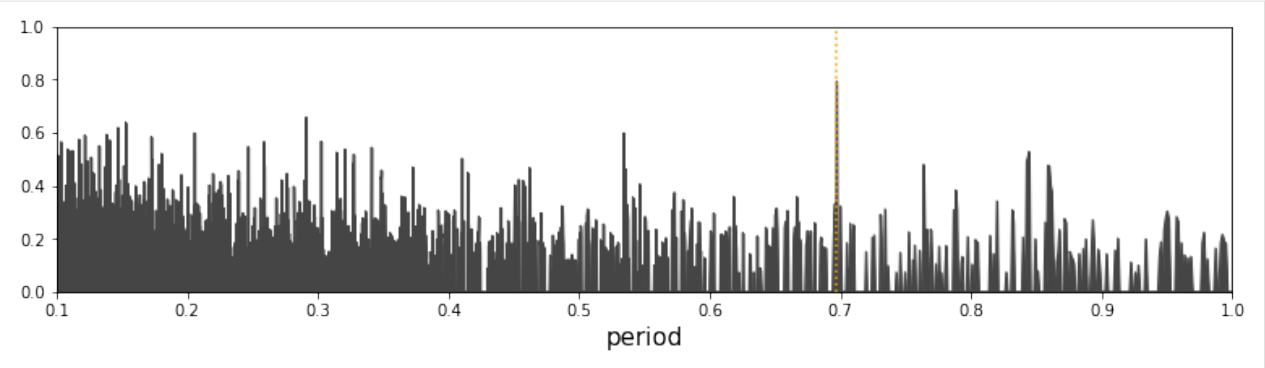
By default, `periodogram` function uses 5-term Fourier model, for which a linear algebra-based faster method is available. For instance, the basic method shown above is equivalent to calling `periodogram(p_min=0.1, p_max=1.0, method='fast', model='Fourier', Nterms=5)`.

Users can change the template model based on the expected shape of light curve. Another pre-implemented function is Gaussian Mixture Model (GMM), which can be specified by changing the `model` argument: `periodogram(p_min=0.1, p_max=1.0, method='custom', model='Gaussian', Nterms=5)`. Since GMM is integrated with Super-Gaussian function in PIPS, users gan give another argument `p`, which changes the power parameter in Super-Gaussian.

Note the change in `method` argument as well: while we implemented Gaussian fitting in log-linear form (`method='fast'`), the resulting fit is often erraneous and thus linear regression (`method='custom'`) is preferred for Gaussian model. More discussion on this topic can be found in our paper.

We internally use `scipy.optimize.curve_fit()` for linear regression. Since this is significantly slower than linear-algebra method, we recommend users to try finding the optimal maximum iteration by changing `maxfev` argument as shown below.

[33]:
```
# periodogram test w/ Gaussian
star.periodogram(p_min=0.1,p_max=1,                ## period search range
                 method='custom',model='Gaussian', ## model selection
                 Nterms=1, p=1,                    ## arguments for the model
                 maxfev=100                        ## max iteration in linear regression
                ).plot(show_peak=True)
```



2.5.4 Using your own custom function

Users can use any custom function as a model to be used in periodogram. The model must be accompanied by initial-guess generator (`p0_func`), both of which needs to take speficic format in the argument. See the function docstrings below.

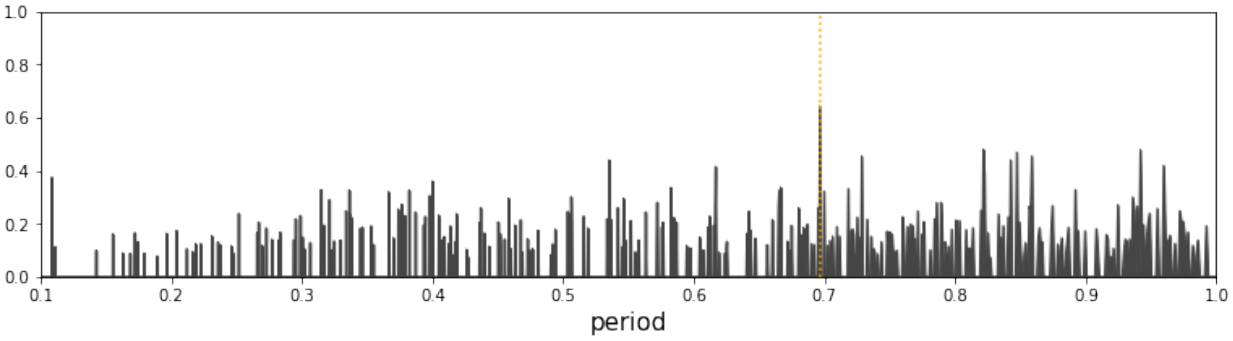
[26]:
```
### define custom functions
from numba import njit
@njit

def polynomial(x,period,params,arg1,arg2=2):
    '''
    An example custom function (model).
    Any custom function must take the arguments (x,period,params),
    and users can add as many fixed (not fitted) arguments as needed.

    In this function, users can define the exponent in the polynomial
    by providing arg1 and arg2.
    '''
    mod = np.remainder(x,period)
    return params[0] + params[1]*(mod-params[3])**arg1 + params[2]*(mod-params[4])**arg2

def poly_p0(x,y,yerr,period,**kwargs):
    '''
    An example of initial-guess generator (p0_func).
    Any p0_func must take the argments (x,y,yerr,period,**kwargs).
    The output array or list must be in the same shape as "params" in the model functon.
    '''
    return [np.mean(y),1,1,period/2,period/2]
```

[32]:
```
### generate periodogram with the custom function
star.periodogram(
                p_min=0.1, p_max=1,   ## period search between 0.1 to 1 day
                method  ='custom',    ## for any custom function this argument needs to␣
↪be given
                model   = polynomial, ## now you can pass the function itself!
                p0_func = poly_p0,    ## initial-guess generator function must be given␣
↪as well
                arg1    = 1,          ## users MUST specify arguments if not default is␣
```

(continues on next page)

(continued from previous page)

```
↪speficied
                arg2    = 4,          ## since arg2=2 is speficified by default, this is␣
↪optional
                maxfev  = 100         ## start with small maxfev and increase later
                ).plot(show_peak=True)
```



2.6 Period detection

Period detection function utilizes `periodogram()` and automatically detects the peak. The periodogram is then refined near the detected peak for accurate period detection. This is followed by linear regression to estimate the uncertainty of detected period.

A few things to note: - `photdata.get_period()` function by default uses 5-term Fourier model. - Users can simply run the function without any arguments to search period between 0.1-4.0 (days). - Detected period and period error is stored in `photdata.period` and `photdata.period_err`. - This function also returns period and period error.

2.6.1 Basic method

[34]:
```
star.get_period(); # no argument -> 5-term Fourier, searches period between 0.1-4 day
print(star.period, star.period_err)
```

```
warning: provided uncertainty may not be accurate. Try increasing sampling size (N0,␣
↪default 10).
0.6969666189351794 2.2946613187290403e-05
```

[8]:
```
period,period_err = star.get_period(p_min=0.1,p_max=1,debug=True) # debug option enables␣
↪the progress printing
print(period,period_err)
```

```
0.000s --- starting the process...
0.000s --- preparing data...
0.000s --- getting a periodogram...
0.509s --- detecting top 5 peaks...
0.510s --- preparing for finer sampling near peaks...
0.511s --- performing finer sampling near peaks...
0.916s --- period candidate:  0.6968767193610299
0.930s --- detecting aliasing...
```

(continues on next page)

(continued from previous page)

```
0.930s --- alias factor:  1
0.931s --- period candidate:  0.6968767193610299
0.932s --- estimating the uncertainty...
0.947s --- period candidate:  0.6968767193610299
0.947s --- period fitted*:  0.6968786839335414
0.947s --- period error:  2.2667570909410562e-05
0.947s --- refining samples...
0.948s --- refining search width = 6.588e-04
1.315s --- period candidate:  0.6968899220719549
1.316s --- period fitted*:  0.6968946264691298
1.316s --- period error:  2.285551532900411e-05
1.316s --- * validating period error...
1.316s --- * fitted period - peak period = 4.70e-06
1.316s --- * expected deviation size = 2.29e-05
1.316s --- * period error validated
1.316s --- period = 0.696890 +- 0.000023d
1.316s --- process completed.
0.6968899220719549 2.285551532900411e-05
```

2.6.2 Advanced method

Since `get_period()` internally calls `periodogram()` function, any arguments that change the setting for `periodogram()` can be applied. For example, users can change the model:

[35]:
```
star.get_period(p_min=0.1,p_max=1.0,method='custom',model='Gaussian')
```

```
---------------------------------------------------------------------------
RemoteTraceback                           Traceback (most recent call last)
RemoteTraceback:
"""
Traceback (most recent call last):
  File "/home/sterlingym/anaconda3/envs/base2/lib/python3.7/multiprocessing/pool.py",␣
↪line 121, in worker
    result = (True, func(*args, **kwds))
  File "/home/sterlingym/anaconda3/envs/base2/lib/python3.7/multiprocessing/pool.py",␣
↪line 44, in mapstar
    return list(map(*args))
  File "/home/sterlingym/Dropbox/projects/PIPS/PIPS/periodogram/custom/custom.py", line␣
↪146, in mp_worker
    return REPRs[repr_mode](MODEL=MODEL,p0_func=P0_FUNC,x=x,y=y,yerr=yerr,period=period,
↪**KWARGS)
KeyError: 'loglik'
"""

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-35-de8f89e3b3fc> in <module>
----> 1 star.get_period(p_min=0.1,p_max=1.0,method='custom',model='Gaussian')

~/Dropbox/projects/PIPS/PIPS/class_photdata/__init__.py in get_period(self, repr_mode,␣
```

(continues on next page)

(continued from previous page)

```
↪return_Z, **kwargs)
    405             return self._get_period(**kwargs)
    406         if repr_mode in ['likelihood','lik','log-likelihood','loglik']:
--> 407             period,period_err,Z = self._get_period_likelihood(repr_mode=repr_
↪mode,**kwargs)
    408             if return_Z:
    409                 return period,period_err,Z

~/Dropbox/projects/PIPS/PIPS/class_photdata/__init__.py in _get_period_likelihood(self,␣
↪period, period_err, p_min, p_max, N_peak, N_noise, Nsigma_range, return_SDE, repr_mode,
↪ **kwargs)
    596             repr_mode='loglik',
    597             raise_warnings=False,
--> 598             **kwargs
    599             )
    600         popt,_ = curve_fit(log_Gaussian,periods,lik,p0=[period,period_err,lik.
↪max()],bounds=[[0,0,-np.inf],[np.inf,np.inf,np.inf]])

~/Dropbox/projects/PIPS/PIPS/periodogram/__init__.py in __call__(self, **kwargs)
     15
     16     def __call__(self,**kwargs):
---> 17         periods,power = self._periodogram(**kwargs)
     18         self.periods = periods
     19         self.power = power

~/Dropbox/projects/PIPS/PIPS/periodogram/__init__.py in _periodogram(self, p_min, p_max,␣
↪custom_periods, N, method, x, y, yerr, plot, multiprocessing, N0, model, raise_
↪warnings, **kwargs)
     91
     92         # main
---> 93         periods,power = METHODS[method](**kwargs)
     94
     95         if 'repr_mode' in kwargs:

~/Dropbox/projects/PIPS/PIPS/periodogram/custom/custom.py in periodogram_custom(x, y,␣
↪yerr, p_min, p_max, N, p0_func, multiprocessing, model, custom_periods, repr_mode,␣
↪**kwargs)
    148     if multiprocessing==True:
    149         pool = Pool()
--> 150         chi2 = pool.map(mp_worker,periods)
    151         pool.close()
    152         pool.join()

~/anaconda3/envs/base2/lib/python3.7/multiprocessing/pool.py in map(self, func, iterable,
↪ chunksize)
    266         in a list that is returned.
    267         '''
--> 268         return self._map_async(func, iterable, mapstar, chunksize).get()
    269
    270     def starmap(self, func, iterable, chunksize=None):

~/anaconda3/envs/base2/lib/python3.7/multiprocessing/pool.py in get(self, timeout)
```

(continues on next page)

(continued from previous page)

```
    655             return self._value
    656         else:
--> 657             raise self._value
    658
    659     def _set(self, i, obj):

KeyError: 'loglik'
```

Similarly, any custom model can be implemented:

[7]:
```
star.get_period(p_min=0.1, p_max=1.0,
                method='custom',
                model=polynomial,
                p0_func=poly_p0,
                arg1 = 1,
                arg2 = 4,
                multiprocessing=False)
```

```
warning: provided uncertainty may not be accurate. Try increasing sampling size (N_peak_
↪test, default 500) and/or turn on the force_refine option.
```

[7]:
```
(0.6969893983974487, 1.8461033093986469e-09)
```

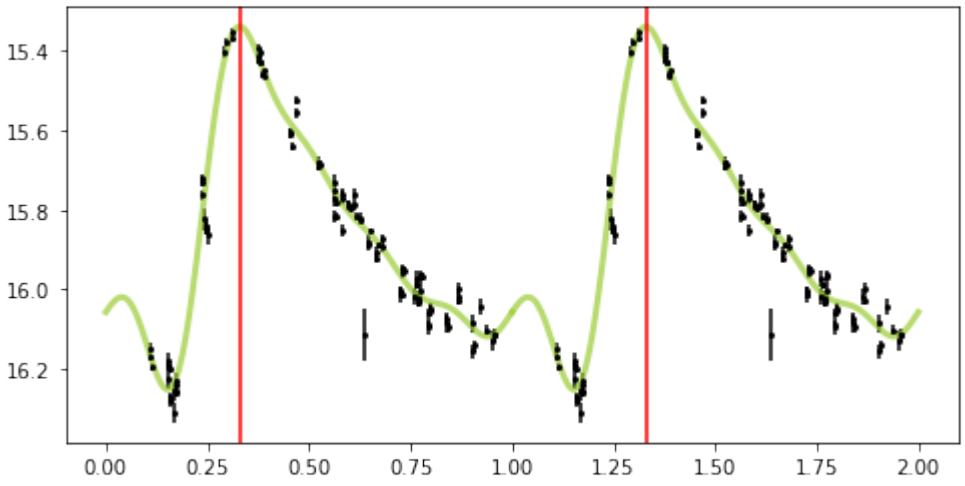
2.7 Visualization

PIPS provides a tool for easy plotting with `plot_lc()`. This function automatically uses the most recently updated period value in `photdata` and returns phase-folded data. There is also an easy way to overplot the best-fit model at the period using `get_bestfit_curve()` function. Like many other functions in `photdata`, users can specify the model and other parameters.

In addition, `get_epoch_offset()` returns the time of maxima offset in phase-folded data (in units of original x-axis: not normalized to unitless phase) and enables easy offsetting / visualization of epoch.

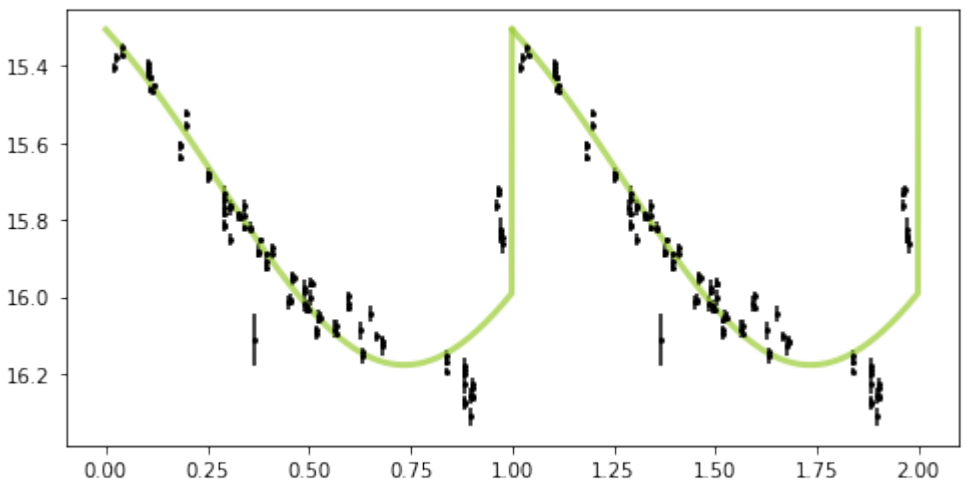
[19]:
```
# detect period
star.get_period()

# phase-folded plot
star.plot_lc() # plots (x%period, y) scatter: normalized to phase
x_th,y_th = star.get_bestfit_curve()
epoch_offset = star.get_epoch_offset() # the epoch offset in the unit of [days] (not␣
↪normalized to phase)

# plot
plt.plot(x_th/star.period,y_th,c='yellowgreen',lw=3,alpha=0.7)
plt.plot(x_th/star.period+1,y_th,c='yellowgreen',lw=3,alpha=0.7)
plt.axvline(epoch_offset/star.period,color='red')
plt.axvline(epoch_offset/star.period+1,color='red')
```

[19]:
```
<matplotlib.lines.Line2D at 0x7f52e36b1dd0>
```



[18]:
```
# get period with Gaussian model
p_Gaussian,p_err_Gaussian = star.get_period(p_min=0.1,p_max=1.0,method='custom',model=
↪'Gaussian')

# auto plot at specified period
star.plot_lc(period=p_Gaussian)
x_th,y_th = star.get_bestfit_curve(period=p_Gaussian,model='Gaussian',Nterms=1,p=1,
↪maxfev=1000000)

# plot
plt.plot(x_th/star.period,y_th,c='yellowgreen',lw=3,alpha=0.7)
plt.plot(x_th/star.period+1,y_th,c='yellowgreen',lw=3,alpha=0.7)
```

[18]:
```
[<matplotlib.lines.Line2D at 0x7f52e33e5f50>]
```



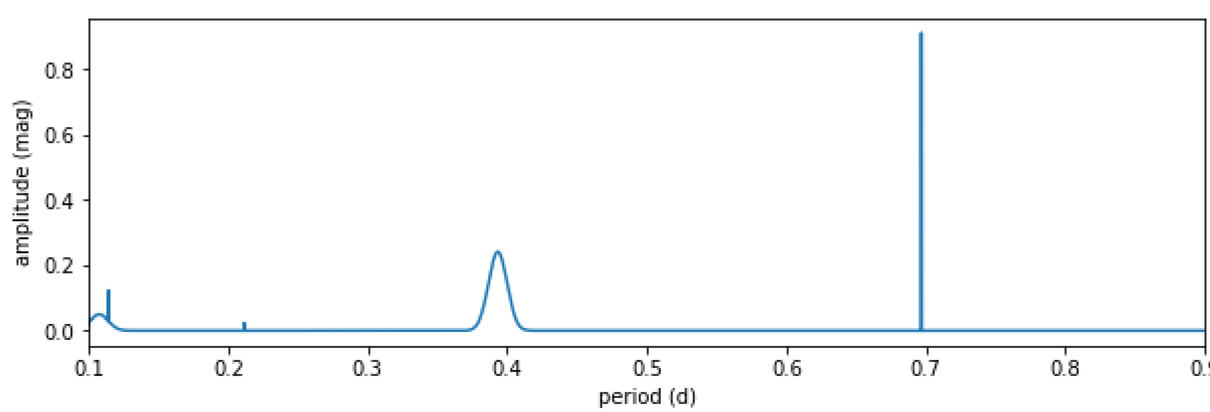
2.8 Multi-period detection

When the object is expected to have more than one period (e.g., double-mode pulsator or variable binaries), the light curve can be a superposition of periodic variation at two or more periods. PIPS can automatically generate the amplitude spectrum of multi-periodic objects.

When `get_period_multi` is called, it returns the detected period and amplitude of top `N` periods. `amplitude_spectrum` internally calls it and generates the amplitude spectrum. It should be noted that, however, PIPS forces the detection, even if the signal is just one of the tallest spikes in the background noise and not the real period.

[8]:
```
# multi-period detection
period,spectrum = star.amplitude_spectrum(p_min=0.1,p_max=0.9,N=10,multiprocessing=False)

plt.figure(figsize=(10,3))
plt.plot(period,spectrum)
plt.xlim(0.1,0.9)
plt.xlabel('period (d)')
plt.ylabel('amplitude (mag)')
plt.show()
```

```
warning: provided uncertainty may not be accurate. Try increasing sampling size (N_peak_
↪test, default 500) and/or turn on the force_refine option.
warning: error size infinity: replacing with periodogram peak width
warning: provided uncertainty may not be accurate. Try increasing sampling size (N_peak_
↪test, default 500) and/or turn on the force_refine option.
warning: error size infinity: replacing with periodogram peak width
warning: provided uncertainty may not be accurate. Try increasing sampling size (N_peak_
↪test, default 500) and/or turn on the force_refine option.
warning: error size infinity: replacing with periodogram peak width
warning: provided uncertainty may not be accurate. Try increasing sampling size (N_peak_
↪test, default 500) and/or turn on the force_refine option.
warning: error size infinity: replacing with periodogram peak width
warning: provided uncertainty may not be accurate. Try increasing sampling size (N_peak_
↪test, default 500) and/or turn on the force_refine option.
warning: error size infinity: replacing with periodogram peak width
```



[ ]:

This notebook is available at https://github.com/SterlingYM/PIPS/tree/master/docs/data_manip.ipynb

CHAPTER

THREE

DATA PREPARATION AND MANIPULATION

[1]:
```
import PIPS
import matplotlib.pyplot as plt
import time
import numpy as np
```

[3]:
```
PIPS.about()
```

```
--------------------------
-    Welcome to PIPS!    -
--------------------------
Version: 0.3.0-beta.1
Authors: Y. Murakami, A. Savel, J. Sunseri, A. Hoffman, Ivan Altunin, Nachiket Girish
--------------------------
Download the latest version from: https://pypi.org/project/astroPIPS
Report issues to: https://github.com/SterlingYM/astroPIPS
Read the documentations at: https://PIPS.readthedocs.io
--------------------------
```

The `photdata` object in PIPS provides tools to perform basic operation for data manipulation. The implemented operation in the current version includes the following:

- Initialization
- Basic info / note storage
- Basic information check (`print()`, `len()`, etc.)
- Data identity check (are two `photdata` objects identical?)
- Data cut (remove data points with uncertainty above threshold)
- Data cut reset
- Data concatenation

3.1 Initialization

The main function of `photdata` is to hold the photometric data. The object requires a list or array or three elements, `x`, `y`, and `yerr` each containing the same length of the time, magnitude (or flux), and magnitude-uncertainty data.

For convenience, datafile generated by LOSSPhotPipeline can be directly imported using a helper function `data_readin_LPP`.

[3]:
```
data = PIPS.data_readin_LPP('../sample_data/005.dat',filter='V')
x,y,yerr = data
print('data shape:\t',np.array(data).shape)
print('x shape:\t',x.shape)
print('y shape:\t',y.shape)
print('y-error shape:\t',yerr.shape)
```

```
data shape:      (3, 103)
x shape:         (103,)
y shape:         (103,)
y-error shape:   (103,)
```

The most basic method to initialize data is to give `data=[x,y,yerr]` as an argument for `PIPS.photdata()`. The initialization function also accepts `label` and `band` arguments, each of which should be used to store information.

[4]:
```
star  = PIPS.photdata(data) # the most basic way to initialize object
star2 = PIPS.photdata(data,label='Star2',band='V') # same as star1, but with more info
```

3.2 copying photdata object

3.3 Printing information

`photdata` accepts some basic python operations, such as `str()`, `print()`, `len()`, and `hash()`. These can be used to check certain properties of the object.

[5]:
```
# basic information check
print('* Basic information check')
print(star)              # the basic information
print(star2)             # the basic information -- label and filter info is included
print('len():',len(star))         # number of datapoints
print('hash():',hash(star))
```

```
* Basic information check
Photdata : band=None, size=103, period=None
Photdata Star2: band=V, size=103, period=None
len(): 103
hash(): 3349517558981431842
```

3.4 Creating an identical copy

photdata object has methods __copy__() and copy(). These methods by default performs a deepcopy instead of shallow copy (binding) to the original object to avoid possible error.

[15]:
```
star3 = star2.copy()    # creates a copy of star3 (including all attributes)
```

3.5 Checking if two objects are identical

[16]:
```
print('when two objects have the same data, they are considered identical')
print('star == star2:',star == star2)     # two objects are identical (hash-equality)
print('star == star3:',star == star3)     # two objects are identical (hash-equality)

print('\nIf any of the data points are changed, they are not identical anymore')
star3.x[-1] = 0         # manually change the data (don't do this!)
print('star == star3:', star == star3)    # False because data itself is changed
```

```
when two objects have the same data, they are considered identical
star == star2: True
star == star3: True

If any of the data points are changed, they are not identical anymore
star == star3: False
```

Note that, since copy() performs deep copy, star2 is unchanged even after star3 has been changed.

[17]:
```
star2.x[-1] # star2 has not been changed
```

[17]:
```
58805.179271
```

3.6 Applying cuts

The cut() function in photdata object offers a quick and easy way to apply data cuts. It takes in xmin, xmax, ymin, ymax, yerr_min, yerr_max arguments, and any combination of these can be applied simultaneously. Once cuts are applied, the existing cuts are effective even when a user applies another cut on different parameter later.

[18]:
```
period,_ = star.get_period(multiprocessing=False)
```

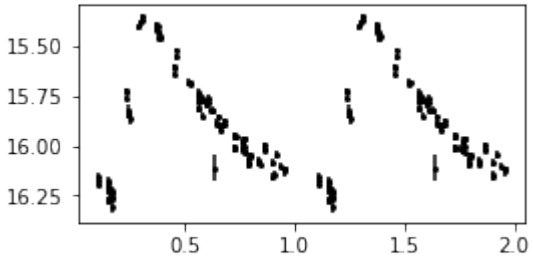
[19]:
```
star2.reset_cuts()
print('Before cuts:', star2)           # info before cuts are applied
star2.plot_lc(period,figsize=(4,2)); plt.show()
star2.cut(yerr_max=0.02)               # cuts data (based on x, y, and yerr values)
print('After a cut:', star2)           # now data is shorter
star2.plot_lc(period,figsize=(4,2)); plt.show()
star2.reset_cuts()
print('After cut reset:', star2)       # reset the cut settings
star2.plot_lc(period,figsize=(4,2)); plt.show()

star2.cut(ymin = 15.8)                 # a cut
```

(continues on next page)

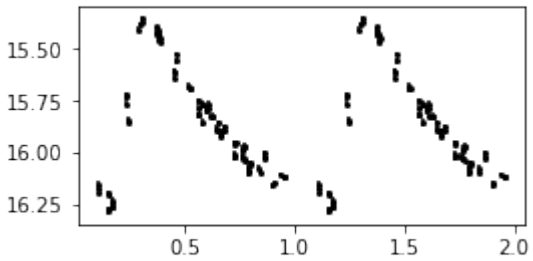
(continued from previous page)

```
star2.cut(yerr_max = 0.01, xmin=58676) # additional cuts
print('After another cut:', star2)     # now both yerr_min and ymin are applied
star2.plot_lc(period,figsize=(4,2)); plt.show()
print('* Even after cuts are applided, two originally identical objects are still␣
↪identical.')
print('star == star2:', star == star2) # raw data is internally preserved
```

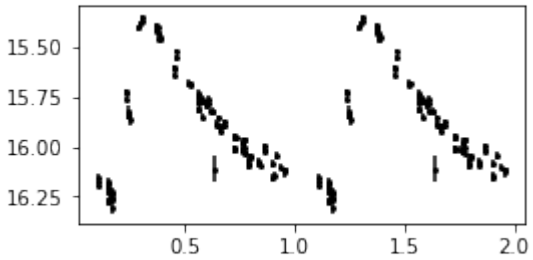
```
Before cuts: Photdata Star2: band=V, size=103, period=None
```



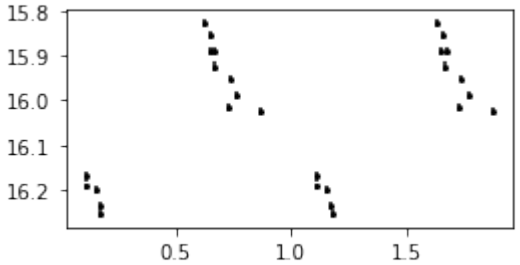

```
After a cut: Photdata Star2: band=V, size=89, period=None
```




```
After cut reset: Photdata Star2: band=V, size=103, period=None
```




```
After another cut: Photdata Star2: band=V, size=14, period=None
```




```
* Even after cuts are applided, two originally identical objects are still identical.
star == star2: True
```

3.7 Combining / concatenating two objects

It is sometimes necessary to merge two sets of data reduced/prepared separately. PIPS offers a quick way to merge two datasets using + (add) operator.

[22]:
```
print(star)
print(star2)
star4 = star + star2     # data concatenation (returns another photdata object)
star4.label = 'Combined' # always add labels
print(star4)             # star4 has more datapoints (notice cuts are applied to star2)
print('star == star4:', star == star4)  # once the data itself is changed, they are not␣
↪identical
```

```
Photdata : band=None, size=103, period=0.6968874975991536
Photdata Star2: band=V, size=14, period=None
Photdata Combined: band=None, size=117, period=None
star == star4: False
```

This notebook is available at https://github.com/SterlingYM/PIPS/tree/master/docs/stellar_parameters.ipynb

CHAPTER

FOUR

STELLAR PARAMETER ESTIMATION USING ASTROPIPS

In this example notebook, we will go through an example of using the `astroPIPS` library to estimate the stellar parameters for RR Lyrae stars. As of version 0.3.0 of `astroPIPS`, we only have empirically derived relationships for stellar parameter estimation applying to RRab and RRc variables.

[1]:
```
# import astroPIPS

import PIPS
```

Now we need to import our variable star data - Import data from a .dat data file - Create a PIPS.photdata object which we call star

[2]:
```
data = PIPS.data_readin_LPP('sample_data/002.dat',filter='V')
x, y, yerr = data
star = PIPS.photdata(data)
```

We now need to estimate the period by running the `get_period` method on our `photdata` object, we choose 5 fourier terms to fit the data with.

[3]:
```
period,err = star.get_period(Nterms=5,multiprocessing=True)

print("The period is " + str(period) + " +/- " + str(err) + " days")
```

```
/Users/jamessunseri/anaconda3/lib/python3.7/site-packages/scipy/optimize/minpack.py:829:␣
↪OptimizeWarning: Covariance of the parameters could not be estimated
  category=OptimizeWarning)
/Users/jamessunseri/anaconda3/lib/python3.7/site-packages/scipy/optimize/minpack.py:829:␣
↪OptimizeWarning: Covariance of the parameters could not be estimated
  category=OptimizeWarning)
```

```
The period is 0.5748866247400108 +/- 7.505536469181548e-06 days
```

If we try to start estimating stellar parameters before calculating the epoch, `astroPIPS` will not be able to properly calculate the stellar parameters. So we need to calculate the epoch offset.

[4]:
```
star.get_epoch_offset()

print("The epoch offset is " + str(star.epoch_offset) + " days.")
```

```
The epoch offset is 0.4212382475572452 days.
```

Now we can calculate stellar parameters! Let's do it. I recommend checking out the methods available first before actually calculating parameters. To use a predicted model, Create a model object from a photdata object. Here we are using the Cacciari2005 model. This model was implemented using equations from Cacciari et al. 2005, citings to all originally derived models are in the docstrings for each parameter estimation method.

[5]:
```
Model = PIPS.Cacciari2005(star)
```

[6]:
```
#We can see all implemented methods available to us by using help(PIPS.Cacciari2005)
print("Here we can see all the different methods for parameter estimation \n\n")
help(PIPS.Cacciari2005)
```

```
Here we can see all the different methods for parameter estimation


Help on class Cacciari2005 in module PIPS.class_StellarModels:

class Cacciari2005(StellarModels)
 |  Cacciari2005(star)
 |
 |  Subclass for StellarModels corresponding to Cacciari's paper from 2005,
 |  this paper is commonly referenced in the literature for RR Lyrae Stellar
 |  Parameter relationships. We denote the original author's relationship's in
 |  the form of method doc strings
 |
 |  Method resolution order:
 |      Cacciari2005
 |      StellarModels
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  calc_BV_0_ab(self)
 |      relationship empirically determined by Kovács & Walker (2001)
 |
 |  calc_Fe_H_ab(self)
 |      relationship derived by Jurscik (1998)
 |
 |  calc_Fe_H_c(self)
 |      relationship derived by Carretta & Gratton (1997)
 |
 |  calc_M_v_ab(self)
 |      relationship derived by Kovács (2002)
 |
 |  calc_M_v_c(self)
 |      relationship derived by Kovács (1998)
 |
 |  calc_all_vals(self, star_type)
 |      "This function returns none. It should add several traits to your StellarModel␣
↪Object:
 |
 |      TRAITS
 |
 |      self.Fe_H
 |      self.BV_0 (RRab only)
 |      self.log_T_eff
 |      self.M_v
 |      self.log_L
 |      self.log_M
```

(continues on next page)

(continued from previous page)

```
|      self.Fe_H_err
|      self.BV_0_err (RRab only)
|      self.log_T_eff_err
|      self.M_v_err
|      self.log_L_err
|      self.log_M_err
|
|  calc_error_BV_0_ab(self)
|
|  calc_error_Fe_H_ab(self)
|
|  calc_error_Fe_H_c(self)
|
|  calc_error_M_v_ab(self)
|
|  calc_error_M_v_c(self)
|
|  calc_error_log_L_ab(self)
|
|  calc_error_log_L_c(self)
|
|  calc_error_log_T_eff_type_ab(self)
|
|  calc_error_log_T_eff_type_c(self)
|
|  calc_error_log_mass_ab(self)
|
|  calc_error_log_mass_c(self)
|
|  calc_error_log_surface_gravity(self)
|
|  calc_log_L_ab(self)
|      Standard luminosity calculation using absolute magnitude.
|      Absolute Magnitude of the Sun reported by Wilmer (2018)
|
|  calc_log_L_c(self)
|      Standard luminosity calculation using absolute magnitude.
|      Absolute Magnitude of the Sun reported by Wilmer (2018)
|
|  calc_log_T_eff_type_ab(self)
|      relationship derived by Kovács & Walker (2001)
|
|  calc_log_T_eff_type_c(self)
|      relationship from Simon & Clement (1993)
|
|  calc_log_mass_ab(self)
|      Originally derived by Jurscik (1998)
|
|  calc_log_mass_c(self)
|      Derived by Simon & Clement (1993)
|
|  calc_log_surface_gravity(self)
```

(continues on next page)

(continued from previous page)

```
|      Cited from Cacciari et al. (2005), standard surface gravity equation
|
|  ----------------------------------------------------------------------
|  Methods inherited from StellarModels:
|
|  __init__(self, star)
|      initialization function for StellarModels Super Class
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from StellarModels:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

Note, one can choose to pick certain stellar properties they want to calculate, by choosing from the list of methods. The typical naming structure is `calc_property_type()` or one can simply calculate all stellar properties with one line.

[7]:
```
#the sample data is of an RRab star
Model.calc_all_vals(star_type='RRab')
```

Similarly we can run `help()` again to see what properties have been calculated for our `Model` object. These properties are stored as object traits to the `Model` object.

[8]:
```
help(Model.calc_all_vals)
```

```
Help on method calc_all_vals in module PIPS.class_StellarModels:

calc_all_vals(star_type) method of PIPS.class_StellarModels.Cacciari2005 instance
    "This function returns none. It should add several traits to your StellarModel␣
↪Object:

    TRAITS

    self.Fe_H
    self.BV_0 (RRab only)
    self.log_T_eff
    self.M_v
    self.log_L
    self.log_M
    self.Fe_H_err
    self.BV_0_err (RRab only)
    self.log_T_eff_err
    self.M_v_err
    self.log_L_err
    self.log_M_err
```

Now we can access these traits:

[9]:
```
print("The metallicity of this star is {0:0.5f} +/- {1:0.5f}".format(Model.Fe_H, Model.
↪Fe_H_err))
print("The log(T_eff/[K]) of this star is {0:0.5f} +/- {1:0.5f}".format(Model.log_T_eff,␣
↪Model.log_T_eff_err))
print("The log(L/[L_sun]) of this star is {0:0.5f} +/- {1:0.5f}".format(Model.log_L,␣
↪Model.log_L_err))
print("The log(M/[M_sun]) of this star is {0:0.5f} +/- {1:0.5f}".format(Model.log_M,␣
↪Model.log_M_err))
```

```
The metallicity of this star is -2.35203 +/- 0.12520
The log(T_eff/[K]) of this star is 3.80587 +/- 0.00147
The log(L/[L_sun]) of this star is 1.71139 +/- 0.00489
The log(M/[M_sun]) of this star is -0.16330 +/- 0.01260
```

There we have it. We have succesfully calculated the stellar properties of a RRab variable star. This can be done for an RRc star as well by modifying the `star_type` argument in the `Model.calc_all_vals()`method.

[ ]: