# BayesFlare Documentation

***Release 1.0.0***

**Matthew Pitkin, Daniel Williams**

July 28, 2015

This is the reference manual for BayesFlare, a pythonic system for handling Kepler light curve data, and using Bayesian techniques to identify stellar flares in them.

If you make use of this software for a publication we would be grateful if you cite Pitkin, Williams, Fletcher and Grant, arXiv:1406.1712.

CHAPTER 1

# A Brief Introduction to BayesFlare

BayesFlare was created to provide an automated means of identifying flaring events in light curves released by the Kepler mission. The aim was to provide a technique that was able to identify even weak events by making use of the flare signal shape. This has led to the modern package containing functions to perform Bayesian hypothesis testing comparing the probability of light curves containing flares to that of them containing noise (or non-flare-like) artefacts.

The statistical methods used in BayesFlare owe much to data analysis developments from the field of gravitational wave research; the detection statistic which is used is based on one developed to identify ring-downs signals from neutron stars in gravitational wave detector data [1].

During the development of the analysis a method was found to account for underlying sinusoidal variations in light curve data by including such variations in the signal model, and then analytically marginalising over them. The functions to do this have also been included in the amplitude-marginaliser suite.

## 1.1 References

[1] Clark, Heng, Pitkin and Woan, *PRD*, **043003** (2007), arXiv:gr-qc/0703138

CHAPTER 2

# Tutorial

The BayesFlare package has been designed to make working with Kepler data easier, and searching for flares in the data more straight-forward.

To use the package you first need to import it into your Python script

```
>>> import bayesflare as bf
```

This tutorial assumes that the module has been imported with the name 'bf'.

In order to conduct any meaningful work with this package, we need access to data from the Kepler mission. At present the package only supports the public light curves, but this may change in the future. The data should be stored in the following format:

```
root-dir
   - Q1_public
   - Q2_public
   - ...
   - Q14_public
```

To access the data it's then as simple as

```
>>> client = bf.Loader('./data')
```

assuming that the root-dir is `./data` in this case.

The loader can then be used to search for lightcurves for any star in the Kepler Input Catalogue;

```
>>> client.find(757450)
```

which will return a list of FITS files in the data folder which correspond to KIC 757450. To work with this data we load it into a Lightcurve object

```
>>> curves = client.find(757450)
>>> lightcurve = bf.Lightcurve(curves[0])
```

The `Lightcurve` object handles a large number of processing for the data, including DC offset removal and de-trending (with e.g. the Savitzky-Golay algorithm implementing in `savitzky_golay()`). When data is added to the `Lightcurve` object it is interpolated to remove `NAN` and `INF` values which cause disruption to a number of the detection processes. The various light curves are then assembled into a combined light curve.

In order to conduct analysis on the data we'll need a model to compare it to. For example, a flare model:

```
>>> M = bf.Flare(lightcurve.cts)
```

This will produce a flare model class containing a signal parameter range (for the Gaussian rise time $\tau_g$ and exponential decay time $\tau_e$) and default grid of signal parameters.

The odds ratio (or Bayes factor) for this model versus Gaussian noise, as a function of time and parameter space, is produced with:

```
>>> B = bf.Bayes(lightcurve, M) # create the Bayes class
>>> B.bayes_factors()           # calculate the log likelihood ratio
```

Within the `Bayes` object this odds ratio is contained within the `numpy.ndarray` *lnBmargAmp*. A final odds ratio as a function of time, with the model parameters marginalised over, is produced via

```
>>> O = B.marginalise_full()
```

where `O` will be a new instance of the `Bayes` class in which *lnBmargAmp* is a 1D array of the natural logarithm of the odds ratio.

Much of this functionality, including thresholding the odds ratio for detection purposes, can be found in the `OddsRatioDetector` class.

More involved examples of using the code can be found in the scripts described in *Pre-made Scripts*

CHAPTER 3

# Pre-made Scripts

The BayesFlare package is supplied with a number of pre-made scripts that can be used as examples of how to perform an analysis. Currently these scripts use the analysis parameters used for the search performed in Pitkin, Williams, Fletcher & Grant, arXiv:1406.1712. The scripts in general provide good examples of using the `OddsRatioDetector` class.

## 3.1 `plot_lightcurve.py`

This script will plot a light curve and the associated log odds ratio. The underlying light curve can either be:

- a simulated curve consisting of pure Gaussian noise,
- a simulated curve consisting of Gaussian noise *and* a sinusoidal variation,
- or, a real Kepler light curve.

The light curve (whether simulated or real) can have a simulated flare signal added to it. By default any simulated light curve will have the same length and time steps as Kepler Quarter 1 long cadence data. By default the noise standard deviation in the data is calculated using the `estimate_noise_ps()` method (with *estfrac* = 0.5).

**The default odds ratio calculation assumes a sliding analysis window of 55 time bins and:**

- a signal model consisting of a flare (with $0 < \tau_g \leq 1800$ seconds, $0 < \tau_g \leq 3600$ seconds and $\tau_g \leq \tau_e$) *and* a 4th order polynomial variation,
- a noise model consisting of a 4th order polynomial *or* a polynomial *and* a positive or negative impulse (anywhere within the analysis window length) *or* a polynomial *and* an exponential decay (with $0 < \tau_g \leq 900$ seconds) *or* a polynomial *and* an exponential rise (with $0 < \tau_g \leq 900$).

To see the command line input options for this script use:

```
>>> ./plot_lightcurve --help
```

## 3.2 `plot_spectrum.py`

This script will plot the one-sided power spectrum of a real or simulated light curve. As with plot-light-curve-label the simulated data in Gaussian, but can also contain a sinusoid, and both real or simulated data can have flare signals added to them.

To see the command line input options for this script use:

```
>>> ./plot_spectrum --help
```

## 3.3 `flare_detection_threshold.py`

This script will compute a threshold on the log odds ratio for a given false alarm rate of detections. The threshold can either be calculated using a set of simulated light curves containing Gaussian noise (and also potentially containing sinusoidal variations randomly produced with amplitude and frequency given specified ranges), or a set of real Kepler light curves.

By default any simulated light curve will have the same length and time steps as Kepler Quarter 1 long cadence data. The noise standard deviation in the data will be calculated using the `estimate_noise_tv()` method (with *sigma* = 1.0). The odds ratio calculation assumes

- a signal model consisting of a flare (with $0 < \tau_g \leq 1800$ seconds, $0 < \tau_g \leq 3600$ seconds and $\tau_g \leq \tau_e$) *and* a 4th order polynomial variation,
- a noise model consisting of a 4th order polynomial *or* a polynomial *and* a positive or negative impulse (anywhere within the analysis window length) *or* a polynomial *and* an exponential decay (with $0 < \tau_g \leq 900$ seconds) *or* a polynomial *and* an exponential rise (with $0 < \tau_g \leq 900$).

The required false alarm probability is given as a the percentage probability that a single light curve (with by default the Kepler Quarter 1 long cadence time scales) will contain a false detection.

To see the command line input options for this script use:

```
>>> ./flare_detection_threshold --help
```

## 3.4 `flare_detection_efficiency.py`

This script will calculate the efficiency of detecting flare signals for a given log odds ratio threshold. This is done by adding simulated flare signals (which by default have time scale parameters drawn at random uniformly within the ranges $0 < \tau_g \leq 1800$ seconds, $0 < \tau_g \leq 3600$ seconds where $\tau_g \leq \tau_e$) with a range of signal-to-noise ratios to simulated, or real, data. Simulated data consists of Gaussian noise to which sinusoidal variations can be added.

By default any simulated light curve will have the same length and time steps as Kepler Quarter 1 long cadence data. The noise standard deviation in the data will be calculated using the `estimate_noise_tv()` method (with *sigma* = 1.0). The odds ratio calculation assumes

- a signal model consisting of a flare (with $0 < \tau_g \leq 1800$ seconds, $0 < \tau_g \leq 3600$ seconds and $\tau_g \leq \tau_e$) *and* a 4th order polynomial variation,
- a noise model consisting of a 4th order polynomial *or* a polynomial *and* a positive or negative impulse (anywhere within the analysis window length) *or* a polynomial *and* an exponential decay (with $0 < \tau_g \leq 900$ seconds) *or* a polynomial *and* an exponential rise (with $0 < \tau_g \leq 900$).

To see the command line input options for this script use:

```
>>> ./flare_detection_efficiency --help
```

## 3.5 `kepler_analysis_script.py`

This scripts was used in the analysis of Pitkin, Williams, Fletcher & Grant to automatically detect flares in Kepler Quarter 1 data. The script will get a list of Kepler stars from MAST (this uses functions heavily indebted to those from

[1]) based on effective temperature and surface gravity criteria (for which the defaults are those used in the analysis in [2] with effective temperature less than 5150 and log(g) greater than 4.2). It will initially ignore any Kepler stars for which the condition flag is not 'None' e.g. it will ignore stars with known exoplanets or planetary candidates.

**Other vetos that are used are**

- stars with known periodicities (including secondary periods) of less than two days are vetoed, based on values given in Tables 1 and 2 of [3] and the table in [3].
- stars in eclipsing binaries (that are not covered by the condition flag veto) are vetoed, based on stars given in [4].

The analysis estimates the data noise standard deviation using the `estimate_noise_tv()` method (with *sigma* = 1.0). The odds ratio calculation assumes

- a signal model consisting of a flare (with $0 < \tau_g \leq 1800$ seconds, $0 < \tau_g \leq 3600$ seconds and $\tau_g \leq \tau_e$) *and* a 4th order polynomial variation,
- a noise model consisting of a 4th order polynomial *or* a polynomial *and* a positive or negative impulse (anywhere within the analysis window length) *or* a polynomial *and* an exponential decay (with $0 < \tau_g \leq 900$ seconds) *or* a polynomial *and* an exponential rise (with $0 < \tau_g \leq 900$).

The results (which includes a list of stars containing flare candidates and the times for each of the flares) are returned in a JSON format text file.

## 3.6 `parameter_estimation_example.py`

This script shows an example of how to perform parameter estimation with the code. It sets up some fake data containing Gaussian noise and adds a simulated flare signal to it. It then sets up a grid in the flare parameter space upon which to calculate the posterior probability distribution. This is then marginalised to produce 1D distributions for each parameter.

## 3.7 References

[1] **kplr** - *A Python interface to the Kepler data* (http://dan.iel.fm/kplr/)
[2] Walkowicz *et al*, *AJ*, **141** (2011), arXiv:1008.0853
[3] Reinhold *et al*, *A&A*, **560**, (2013), arXiv:1308.1508
[4] Prsa *et al*, *AJ*, **141** (2011), arXiv:1006.2815

CHAPTER 4

# API Reference

## 4.1 Data Handling

The classes defined here are used to find and store Kepler light curves.

The `Lightcurve` class is in general the required data format for passing to the Bayesian analysis functions.

**Note:** At the moment the `Lightcurve` class is specifically designed for Kepler data (although it could hold any time series if necessary by hardwiring some values). In the future versions it will be made more generic.

### 4.1.1 Data Loader

## 4.2 Models

These classes provide model functions and also set up grid over the parameters spaces of the models and provide log(prior) values for those parameter spaces.

A generic model class is provided that can set up model parameters, parameter ranges, and filtering functions. This class is inherited by all the other models, which provide model functions and priors.

### 4.2.1 The Flare Model

This class provides a flare model in which the flare light curve has a Gaussian rise and an exponential decay as given by m(t, $\tau_g, \tau_e, T_0) = A_0 \begin{cases} e^{-(t-T_0)^2/(2\tau_g^2)} & \text{if } t \leq T_0, \\ e^{-(t-T_0)/\tau_e} & \text{if } t > T_0, \end{cases} where \tau_g$ is the width of the Gaussian rise, $\tau_e$ is the time constant of the exponential decay, $T_0$ is the time of the flare peak, and $A_0$ is the peak amplitude.

In this class the parameter space grid and prior is set up such that $\tau_e > \tau_g$.

### 4.2.2 The Transit Model

This class provides a generic transit model. It is not the fully physical model of [1], but is instead a simple model with Gaussian wings and a flat trough.

The parameter space grid and prior is set up such that the total length on the transit does not exceed a given value.

[1] Mandel and Agol, *Ap. J. Lett.*, **580** (2002), arXiv:astro-ph/0210099.

### 4.2.3 General models

These classes provide models for a range of generic signal types. These can be used either as signal or noise models when forming an odds ratio.

#### Exponential decay/rise

A class giving a model with exponential decay or rise.

#### Impulse

A class giving a model with a delta-function-like (single bin) impulse.

#### Gaussian

A class giving a model with a Gaussian profile.

#### Step

A class giving a step function profile.

### 4.2.4 References

## 4.3 Bayesian Functions

### 4.3.1 The Bayes and Parameter estimation classes

### 4.3.2 Likelihood marginalisation and helper functions

These `C` and Cython functions provide tools, and wrappers, to perform the likelihood ratio evaluation. There are also helper functions for use in numerical integration of functions.

#### `C`-functions

These `C`-functions are used performing likelihood ratio evaluation when analytically marginalising over a set of model component amplitudes. For further documentation of these functions see [2] and [3].

**`LOG2PI`**
The natural logarithm of $2\pi$ defined using the GSL macros `(M_LN2 + M_LNPI)`

**`LOGPI_2`**
The natural logarithm of $\pi/2$ defined using the GSL macros `(M_LNPI - M_LN2)`

[2] http://mattpitkin.github.io/amplitude-marginaliser

[3] Pitkin, Williams, Fletcher and Grant, 2014, arXiv:1406.1712.

double **`log_marg_amp_full_C`**(int *Nmodels*, double *modelModel[]*, double *dataModel[]*, double *sigma*, unsigned int *lastHalfRange*)

This function calculates the log-likelihood ratio for a signal model (consisting of a number *Nmodels* components, each with an amplitude that is marginalised over) compared to a pure Gaussian noise model. For all bar the last model component the amplitudes are analytically marginalised between $-\infty$ and $\infty$, whilst if *lastHalfRange* is true the final amplitude will be marginalised between 0 and infinity (i.e. it must be positive).

For a more complete description of this function see [1] or Algorithm 1 in [2].

**Parameters** **Nmodels** : int

The number of model components.

**modelModel** : double array

A flattened 1D array consisting of the summed cross terms for each model component [size: *Nmodels* -by- *Nmodels*].

**dataModel** : double array

A 1D array of the summed data crossed with each model component [size: *Nmodels*]

**sigma** : double

The underlying Gaussian noise standard deviation

**lastHalfRange** : bool

A boolean saying whether the final model component amplitudes should be marginalised over the full $-\infty$ to $\infty$ range (False), or between 0 and $\infty$ (True).

**Returns** **logL** : double

The log-likelihood ratio marginalised over the model component amplitudes.

double **`log_marg_amp_except_final_C`**(int *Nmodels*, double *modelModel[]*, double *dataModel[]*, double *sigma*)

This function calculates the log-likelihood ratio for a signal model (consisting of a number *Nmodels* components) compared to a pure Gaussian noise model. For all bar the last model component the amplitudes are analytically marginalised between $-\infty$ and $\infty$.

For a more complete description of this function see [1] or Algorithm 2 in [2].

**Parameters** **Nmodels** : int

The number of model components.

**modelModel** : double array

A flattened 1D array consisting of the summed cross terms for each model component [size: *Nmodels* -by- *Nmodels*].

**dataModel** : double array

A 1D array of the summed data crossed with each model component [size: *Nmodels*]

**sigma** : double

The underlying Gaussian noise standard deviation

**Returns** **logL** : double

The log-likelihood ratio marginalised over the model component amplitudes.

#### Cython functions

#### References

## 4.4 Noise Estimation and Simulation

## 4.5 Flare Identification

These classes provide methods to identify flares in light curve data.

## 4.6 Miscellaneous Functions

CHAPTER 5

# Acknowledgements

The development of the BayesFlare module and the scripts for which it was created was part-funded by the Royal Society of Edinburgh Cormack Bequest.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index