shocksgo Documentation

Release 0.0.dev37

Brett Morris

Feb 12, 2019

Contents

This is the documentation for shocksgo. The goal of shocksgo is to generate light curves of stars accounting for the effects of granulation, super-granulation and p-mode oscillations.

You can view the source code and/or contribute to shocksgo via GitHub.

Part I

Documentation

CHAPTER 1

Installing shocksgo

You can install shocksgo with pip:

```
pip install shocksgo
```

You can install shocksgo from the source code by doing the following:

```
git clone https://github.com/bmorris3/shocksgo.git
cd shocksgo
python setup.py install
```

shocksgo requires python >=3.5, numpy, celerite, and astropy.

If you have any trouble installing shocksgo, feel free to post an issue on GitHub.

CHAPTER 2

Getting Started

This tutorial will show some examples of how to make solar and stellar light curves using shocksgo.

2.1 Generating a Solar Light Curve

To generate a sample of ten hours of solar fluxes at 60 second cadence, we can use `generate_solar_fluxes`:

```
import matplotlib.pyplot as plt
import astropy.units as u

from shocksgo import generate_solar_fluxes

times, fluxes, kernel = generate_solar_fluxes(duration=10*u.hour, cadence=60*u.s)

plt.plot(times.to(u.hour), 1e6 * fluxes)
plt.gca().set(xlabel='Time [hours]', ylabel='Relative Flux [ppm]')
plt.show()
```

We can check that the power spectrum of the fluxes that we've generated reproduce the solar power spectrum:

```
import matplotlib.pyplot as plt
import numpy as np
import astropy.units as u

from shocksgo import generate_solar_fluxes, power_spectrum

times, fluxes, kernel = generate_solar_fluxes(duration=100*u.day, cadence=60*u.s)

freq, power = power_spectrum(fluxes, d=60)

plt.loglog(freq * 1e6, power, ',', label='Samples')
plt.loglog(freq * 1e6, 1e6 * kernel.get_psd(2*np.pi*freq)/(2*np.pi), alpha=0.7, label='Kernel')
plt.ylim([1e-5, 1e3])
```

(continues on next page)

(continued from previous page)

```
plt.xlim([1e-2, 1e4])
plt.gca().set(xlabel='Frequency [$\mu$Hz]', ylabel='Power [ppm$^2$/$\mu$Hz]')
plt.show()
```

Zooming into the p-mode oscillations, we can see the peaks are reproduced:

2.2 Generating a Stellar Light Curve

To generate a sample of *steller* fluxes at 60 second cadence, we can use `generate_stellar_fluxes`:

```
import matplotlib.pyplot as plt
import astropy.units as u
from astropy.constants import M_sun, L_sun, R_sun

from shocksgo import generate_stellar_fluxes

# Stellar properties
M = 0.9 * M_sun
T_eff = 5340 * u.K
L = 0.56 * L_sun
R = 0.7 * R_sun

times, fluxes, kernel = generate_stellar_fluxes(duration=100*u.day, M=M, T_eff=T_eff, R=R, L=L,␣
↪cadence=60*u.s)

plt.plot(times.to(u.day), 1e6 * fluxes)
plt.gca().set(xlabel='Time [days]', ylabel='Relative Flux [ppm]', title='G9V star')
plt.show()
```

We can see the shift in the p-mode oscillations relative to the solar ones above if we plot the power spectrum:

```
import matplotlib.pyplot as plt
import numpy as np
import astropy.units as u
from astropy.constants import M_sun, L_sun, R_sun

from shocksgo import generate_stellar_fluxes, power_spectrum

# Stellar properties
M = 0.9 * M_sun
T_eff = 5340 * u.K
L = 0.56 * L_sun
R = 0.876 * R_sun

times, fluxes, kernel = generate_stellar_fluxes(duration=10*u.day, M=M, T_eff=T_eff, R=R, L=L,␣
↪cadence=60*u.s)

freq, power = power_spectrum(fluxes, d=60)

plt.semilogy(freq * 1e6, power, ',', label='Samples')
plt.semilogy(freq * 1e6, 1e6 * kernel.get_psd(2*np.pi*freq)/(2*np.pi), alpha=0.7, label='Kernel')
plt.ylim([1e-5, 1e-1])
plt.xlim([2500, 5000])
plt.gca().set(xlabel='Frequency [$\mu$Hz]', ylabel='Power [ppm$^2$/$\mu$Hz]')
plt.show()
```

2.3 Custom Frequencies

Suppose you have a list of model p-mode frequencies, and you would like to generate a light curve from your custom list of frequencies (without scaling from the solar values). You can do so using a different set of keyword arguments in the `generate_stellar_fluxes` function, like so:

```
import numpy as np
import matplotlib.pyplot as plt
import astropy.units as u
from astropy.constants import R_sun, M_sun, L_sun

from shocksgo import generate_stellar_fluxes

M = 1*M_sun
L = 1*L_sun
T_eff = 5777 * u.K
R = 1*R_sun

freqs = np.linspace(2000, 4000, 10)  # in microHertz
log_amps = np.exp(-0.5 * (freqs - 3000)**2 / 1000**2) - 32
log_lifetimes = np.ones_like(freqs) * 7
duration = 2 * u.min

times, fluxes, kernel = generate_stellar_fluxes(duration, M, T_eff, R, L,
                                                frequencies=freqs,
                                                log_amplitudes=log_amps,
                                                log_mode_lifetimes=log_lifetimes)

test_freqs = np.logspace(-2, np.log10(freqs.max()), 1e6)
plt.loglog(test_freqs, 1e6/(2*np.pi) * kernel.get_psd(2*np.pi*test_freqs*1e-6))
plt.gca().set(xlabel='Frequency [$\mu$Hz]', ylabel='Power [ppm$^2$/$\mu$Hz]')
plt.show()
```

CHAPTER 3

shocksgo Documentation

This is the documentation for shocksgo.

3.1 Reference/API

3.1.1 shocksgo Package

Functions

generate_solar_fluxes(duration[, cadence])	Generate an array of fluxes with zero mean which mimic the power spectrum of the SOHO/VIRGO SPM observations.
generate_stellar_fluxes(duration, M, T_eff, R, L)	Generate an array of fluxes with zero mean which mimic the power spectrum of the SOHO/VIRGO SPM observations, scaled for a star with a given mass, effective temperature, luminosity and radius.
interpolate_missing_data(times, fluxes[, ...])	Assuming `times` are uniformly spaced with missing cadences, fill in the missing cadences with linear interpolation.
power_spectrum(fluxes[, d])	Compute the power spectrum of `fluxes` in units of [ppm^2 / microHz].
test(**kwargs)	Run the tests for the package.

generate_solar_fluxes

shocksgo.**generate_solar_fluxes**(*duration*, *cadence=<Quantity 60. s>*)

Generate an array of fluxes with zero mean which mimic the power spectrum of the SOHO/VIRGO SPM observations.

Parameters

duration
[Quantity] Duration of simulated observations to generate.

cadence
[Quantity] Length of time between fluxes

Returns

times
[Quantity] Array of times at cadence cadence of length duration/cadence

fluxes
[ndarray] Array of fluxes at cadence cadence of length duration/cadence

kernel
[TermSum] Celerite kernel used to approximate the solar power spectrum.

generate_stellar_fluxes

shocksgo.**generate_stellar_fluxes**(*duration*, *M*, *T_eff*, *R*, *L*, *cadence=<Quantity 60. s>*, *frequencies=None*, *log_amplitudes=None*, *log_mode_lifetimes=None*)

Generate an array of fluxes with zero mean which mimic the power spectrum of the SOHO/VIRGO SPM observations, scaled for a star with a given mass, effective temperature, luminosity and radius.

Parameters

duration
[Quantity] Duration of simulated observations to generate.

M
[Quantity] Stellar mass

T_eff
[Quantity] Stellar effective temperature

R
[Quantity] Stellar radius

L
[Quantity] Stellar luminosity

cadence
[Quantity] Length of time between fluxes

frequencies
[ndarray or None] p-mode frequencies in the power spectrum in units of microHertz. Defaults to scaled solar values.

log_amplitudes
[ndarray or None] p-mode amplitudes in the power spectrum. Defaults to scaled solar values.

log_mode_lifetimes
[ndarray or None] p-mode lifetimes in the power spectrum. Defaults to scaled solar values.

Returns

times
[Quantity] Array of times at cadence cadence of size duration/cadence

fluxes
[ndarray] Array of fluxes at cadence cadence of size duration/cadence

kernel
[TermSum] Celerite kernel used to approximate the stellar power spectrum

interpolate_missing_data

shocksgo.**interpolate_missing_data**(*times*, *fluxes*, *cadences=None*)
Assuming times are uniformly spaced with missing cadences, fill in the missing cadences with linear interpolation.

Cadences can be passed if they are known.

Parameters

times
[numpy.ndarray] Incomplete but otherwise uniformly sampled times

fluxes
[numpy.ndarray] Flux for each time in times

cadences
[numpy.ndarray, optional] Integer cadence number of each observation.

Returns

interpolated_times
[numpy.ndarray] times with filled-in missing cadences

interpolated_fluxes
[numpy.ndarray] fluxes with filled-in missing cadences

power_spectrum

shocksgo.**power_spectrum**(*fluxes*, *d=60*)
Compute the power spectrum of fluxes in units of [ppm^2 / microHz].

Parameters

fluxes
[ndarray] Fluxes with zero mean.

d
[float] Time between samples [s].

Returns

freq
[ndarray] Frequencies

power
[ndarray] Power at each frequency in units of [ppm^2 / microHz]

test

shocksgo.**test**(***kwargs*)

Run the tests for the package.

This method builds arguments for and then calls `pytest.main`.

Parameters

package
[str, optional] The name of a specific package to test, e.g. 'io.fits' or 'utils'. Accepts comma separated string to specify multiple packages. If nothing is specified all default tests are run.

args
[str, optional] Additional arguments to be passed to `pytest.main` in the `args` keyword argument.

docs_path
[str, optional] The path to the documentation .rst files.

open_files
[bool, optional] Fail when any tests leave files open. Off by default, because this adds extra run time to the test suite. Requires the `psutil` package.

parallel
[int or 'auto', optional] When provided, run the tests in parallel on the specified number of CPUs. If parallel is `'auto'`, it will use the all the cores on the machine. Requires the `pytest-xdist` plugin.

pastebin
[('failed', 'all', None), optional] Convenience option for turning on py.test pastebin output. Set to 'failed' to upload info for failed tests, or 'all' to upload info for all tests.

pdb
[bool, optional] Turn on PDB post-mortem analysis for failing tests. Same as specifying `--pdb` in `args`.

pep8
[bool, optional] Turn on PEP8 checking via the pytest-pep8 plugin and disable normal tests. Same as specifying `--pep8 -k pep8` in `args`.

plugins
[list, optional] Plugins to be passed to `pytest.main` in the `plugins` keyword argument.

remote_data
[{'none', 'astropy', 'any'}, optional] Controls whether to run tests marked with @pytest.mark.remote_data. This can be set to run no tests with remote data (`none`), only ones that use data from http://data.astropy.org (`astropy`), or all tests that use remote data (`any`). The default is `none`.

repeat
[`int`, optional] If set, specifies how many times each test should be run. This is useful for diagnosing sporadic failures.

skip_docs
[`bool`, optional] When `True`, skips running the doctests in the .rst files.

test_path
[str, optional] Specify location to test by path. May be a single file or directory. Must be specified absolutely or relative to the calling directory.

verbose
: [bool, optional] Convenience option to turn on verbose output from py.test. Passing True is the same as specifying `-v` in `args`.

Classes

UnsupportedPythonError

UnsupportedPythonError

exception shocksgo.**UnsupportedPythonError**

Class Inheritance Diagram

Part II

Overview

CHAPTER 4

Methods

We compute these light curves efficiently by taking advantage of celerite, a fast Gaussian process regression package, which we use to approximate solar and stellar power spectrum densities with sums of simple harmonic oscillator (SHO) kernels of the form:

$$S(\omega) = \sqrt{\frac{2}{\pi}} \frac{S_0\,\omega_0^4}{(\omega^2 - {\omega_0}^2)^2 + {\omega_0}^2\,\omega^2/Q^2}$$

where $\omega = 2\pi f$ is the angular frequency. We use one SHO kernel term for super/meso-granulation, another for ordinary granulation, and about 50 terms for the comb of p-mode peaks.

4.1 Scaling relations for p-modes

For computation of stellar p-mode oscillation frequencies, we use the scaling relations found in Huber et al. (2012) and references therein (e.g. Kjeldsen & Bedding 1995), namely Equation 4:

$$\nu_{\rm max} \propto M R^{-2} T_{\rm eff}^{-1/2},$$

and Equation 3

$$\Delta\nu_{\rm max} \propto M^{1/2} R^{-3/2}.$$

The amplitude scaling of the p-mode oscillations is given by Equation 9 of Huber et al. (2011):

$$A \propto \frac{L^s}{M^t T_{\rm eff}^{r-1} c(T_{\rm eff})}$$

where $r = 2$, $s = 0.886$, $t = 1.89$ and

$$c(T_{\rm eff}) = \left(\frac{T_{\rm eff}}{5934{\rm K}}\right)^{0.8}.$$

4.2 Scaling relations for granulation

For computation of large and small scale stellar surface granulation frequencies, we use the scaling relation found in Kallinger et al. (2014):

$$\tau_{\rm eff} \propto \nu_{\rm max}^{-0.89},$$

where $\tau_{\rm eff}$ is the characteristic granulation timescale. The amplitudes of granulation scale as

$$a \propto \nu_{\rm max}^{-2}.$$

(Kjeldsen & Bedding, 2011).

Python Module Index

S

Index