cuvarbase Documentation

Release 0.2.4

John Hoffman

Jul 13, 2022

Contents:

`cuvarbase` is a Python library that uses PyCUDA to implement several time series tools used in astronomy on GPUs.

See the documentation.

This project is under active development, and currently includes implementations of

- Generalized Lomb Scargle periodogram
- Box-least squares (BLS)
- Non-equispaced fast Fourier transform (adjoint operation) (NFFT paper)
- Conditional entropy period finder (CE)
- **Phase dispersion minimization (PDM2)**
  - Currently operational but minimal unit testing or documentation (yet)

Hopefully future developments will have

- (Weighted) wavelet transforms
- Spectrograms (for PDM and GLS)
- Multiharmonic extensions for GLS

CHAPTER 1

Dependencies

- PyCUDA **<-essential**
- **scikit cuda <-also essential**
  - used for access to the CUDA FFT runtime library
- matplotlib (for plotting utilities)
- nfft (for unit testing)
- astropy (for unit testing)

CHAPTER 2

Using multiple GPUs

If you have more than one GPU, you can choose which one to use in a given script by setting the `CUDA_DEVICE` environment variable:

```
CUDA_DEVICE=1 python script.py
```

If anyone is interested in implementing multi-device load-balancing solution, they are encouraged to do so! At some point this may become important, but for the time being manually splitting up the jobs to different GPU's will have to suffice.

2.1 What's new in cuvarbase

- **0.2.4**
  - bugfix for pytest (broke b/c of incorrect fixture usage)
  - added `ignore_negative_delta_sols` option to BLS to ignore inverted dips in the lightcurve
- **0.2.1**
  - bugfix for memory leak in BLS
  - contact email changed in setup
- **0.2.0**
  - Many more unit tests for BLS and CE.
  - **BLS**
    - Now several orders of magnitude faster! Use `use_fast=True` in `eebls_transit_gpu` or use `eebls_gpu_fast`.
    - Bug-fix for boost-python error when calling `eebls_gpu_fast`.
  - **CE**

    * New `use_fast` parameter in `ConditionalEntropyAsyncProcess`; if selected will use a kernel that should be substantially more efficient and that requires no memory overhead. If selected, you should use the `run` function and not the `large_run` function. Currently the `weighted` option is not supported when `use_fast` is `True`.
    * Bug-fix for `mag_overlap > 0`.

- **0.1.9**
  - Added Sphinx documentation
  - **Now Python 3 compatible!**
  - Miscillaneous bug fixes
  - **CE**
    * Run functions for `ConditionalEntropyAsyncProcess` now allow for a `balanced_magbins` argument to set the magnitude bins to have widths that vary with the distribution of magnitude values. This is more robust to outliers, but performance comparisons between the usual CE algorithm indicate that you should use care.
    * Added `precompute` function to `ConditionalEntropyAsyncProcess` that allows you to speed up computations without resorting to the `batched_run_constant_nfreq` function. Currently it still assumes that the frequencies used will be the same for all lightcurves.
  - **GLS**
    * Added `precompute` function to `LombScargleAsyncProcess`.
    * Avoids allocating GPU memory for NFFT when `use_fft` is `False`.
    * `LombScargleAsyncProcess.memory_requirement` is now implemented.
  - **BLS**
    * `eebls_gpu`, `eebls_transit_gpu`, and `eebls_custom_gpu` now have a `max_memory` option that allows you to automatically set the `batch_size` without worrying about memory allocation errors.
    * `eebls_transit_gpu` now allows for a `freqs` argument and a `qvals` argument for customizing the frequencies and the fiducial `q` values
    * Fixed a small bug in `fmin_transit` that miscalculated the minimum frequency.
- **0.1.8**
  - Removed gamma function usage from baluev 2008 false alarm probability (`use_gamma=True` will override this)
  - Fixed a bug in the GLS notebook
- **0.1.6/0.1.7**
  - Some bug fixes for GLS
  - `large_run` function for Conditional Entropy period finder allows large frequency grids without raising memory allocation errors.
  - More unit tests for conditional entropy
  - Conditional entropy now supports double precision with the `use_double` argument
- **0.1.5**
  - **Conditional Entropy period finder now unit tested**

        - Weighted variant also implemented – accounts for heteroskedasticity if that's important

  - **BLS**
    - New unit tests
    - **A new transiting exoplanet BLS function: `eebls_transit_gpu`**
      - Only searches plausible parameter space for Keplerian orbit
  - **GLS**
    - **False alarm probability: `fap_baluev`**
      - Implements Baluev 2008 false alarm probability measure based on extreme value theory

2.2 Install instructions

These installation instructions are for Linux/BSD-based systems (OS X/macOS, Ubuntu, etc.). Windows users, your suggestions and feedback is welcome if we can make your life easier!

2.2.1 Installing the Nvidia Toolkit

`cuvarbase` requires PyCUDA and scikit-cuda, which both require the Nvidia toolkit for access to the Nvidia compiler, drivers, and runtime libraries.

Go to the NVIDIA Download page and select the distribution for your operating system. Everything has been developed and tested using **version 8.0**, so it may be best to stick with that version for now until we verify that later versions are OK.

Warning: Make sure that your `$PATH` environment variable contains the location of the `CUDA` binaries. You can test this by trying `which nvcc` from your terminal. If nothing is printed, you'll have to amend your `~/.bashrc` file:

```
echo "export PATH=/usr/local/cuda/bin:${PATH}" >> ~/.bashrc && . ~/.bashrc
```

The `>>` is not a typo – using one `>` will *overwrite* the `~/.bashrc` file. Make sure you change `/usr/local/cuda` to the appropriate location of your Nvidia install.

Also important

Make sure your `$LD_LIBRARY_PATH` and `$DYLD_LIBRARY_PATH` are also similarly modified to include the `/lib` directory of the CUDA install:

```
echo "export LD_LIBRARY_PATH=/usr/local/cuda/lib:${LD_LIBRARY_PATH}" >>
~/.bashrc && . ~/.bashrc     echo "export DYLD_LIBRARY_PATH=/usr/local/cuda/
lib:${DYLD_LIBRARY_PATH}" >> ~/.bashrc && . ~/.bashrc
```

2.2.2 Using conda

Conda is a great way to do this in a safe, isolated environment.

First create a new conda environment (named `pycu` here) that will use Python 2.7 (python 2.7, 3.4, 3.5, and 3.6 have been tested), with the numpy library installed.

```
conda create -n pycu python=2.7 numpy
```

Note: The numpy library *has* to be installed *before* PyCUDA is installed with pip. The PyCUDA setup needs to be able to access the numpy library for building against it. You can do this with the above command, or alternatively just do `pip install numpy && pip install cuvarbase`

Then activate the virtual environment

```
source activate pycu
```

and then use `pip` to install `cuvarbase`

```
pip install cuvarbase
```

2.2.3 Installing with just `pip`

If you don't want to use conda the following should work with just pip

```
pip install numpy
pip install cuvarbase
```

2.2.4 Troubleshooting PyCUDA installation problems

The `PyCUDA` installation step may be a hiccup in this otherwise orderly process. If you run into problems installing `PyCUDA` with pip, you may have to install PyCUDA from source yourself. It's not too bad, but if you experience any problems, please submit an Issue at the `cuvarbase` Github page and I'll amend this documentation.

Below is a small bash script that (hopefully) automates the process of installing PyCUDA in the event of any problems you've encountered at this point.

```
PYCUDA="pycuda-2017.1.1"
PYCUDA_URL="https://pypi.python.org/packages/b3/30/
↪9e1c0a4c10e90b4c59ca7aa3c518e96f37aabcac73ffe6b5d9658f6ef843/pycuda-2017.1.1.tar.gz
↪#md5=9e509f53a23e062b31049eb8220b2e3d"
CUDA_ROOT=/usr/local/cuda

# Download
wget $PYCUDA_URL

# Unpack
tar xvf ${PYCUDA}.tar.gz
cd $PYCUDA

# Configure with current python exe
./configure.py --python-exe=`which python` --cuda-root=$CUDA_ROOT
python setup.py build
python setup.py install
```

If everything goes smoothly, you should now test if `pycuda` is working correctly.

```
python -c "import pycuda.autoinit; print 'Hurray!'"
```

If everything works up until now, we should be ready to install `cuvarbase`

```
pip install cuvarbase
```

2.2.5 Installing from source

You can also install directly from the repository. Clone the `git` repository on your machine:

```
git clone https://github.com/johnh2o2/cuvarbase
```

Then install!

```
cd cuvarbase
python setup.py install
```

The last command can also be done with pip:

```
pip install -e .
```

2.2.6 Troubleshooting on a Mac

Nvidia offers CUDA for Mac OSX. After installing the package via downloading and running the `.dmg` file, you'll have to make a couple of edits to your `~/.bash_profile`:

```
export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH}:/usr/local/cuda/lib"
export PATH="/usr/local/cuda/bin:${PATH}"
```

and then source these changes in your current shell by running `. ~/.bash_profile`.

Another important note: **nvcc (8.0.61) does not appear to support the latest clang compiler**. If this is the case, running `python example.py` should produce the following error:

```
nvcc fatal   : The version ('80100') of the host compiler ('Apple clang') is not␣
↪supported
```

You can fix this problem by temporarily downgrading your clang compiler. To do this:

- Download Xcode command line tools 7.3.1
- Install.
- Run `sudo xcode-select --switch /Library/Developer/CommandLineTools` until `clang --version` says `7.3`.

2.3 Conditional Entropy

The conditional entropy period finder [G2013] phase-folds the data at each trial frequencies and estimates the conditional entropy $H(m|\phi)$ of the data. The idea is that the data with the least entropy (intuitively: the greatest "structure" or "non-randomness"), should correspond to the correct frequency of a stationary signal.

Here,

$$H(m|\phi) = H(m, \phi) - H(\phi) = \sum_{m,\phi} p(m, \phi) \log \left(\frac{p(\phi)}{p(m, \phi)} \right)$$

where $p(m, \phi)$ is the density of points that fall within the bin located at phase ϕ and magnitude m and $p(\phi) = \sum_m p(m, \phi)$ is the density of points that fall within the phi range.

2.3.1 An example with `cuvarbase`

```python
import cuvarbase.ce as ce
import numpy as np

# make some fake data
t = np.sort(np.random.rand(100))
y = np.cos(2 * np.pi * 10 * t)
y += np.random.randn(len(t))
dy = np.ones_like(t)

# start a conditional entropy process
proc = ConditionalEntropyAsyncProcess(phase_bins=10, mag_bins=5)

# format your data as a list of lightcurves (t, y, dy)
data = [(t, y, dy)]

# run the CE process with your data
results = proc.run(data)

# finish the process (probably not necessary but ensures
# all data has been transferred)
proc.finish()

# Results is a list of [(freqs, CE), ...] for each lightcurve
# in ``data``.
freqs, ce_spectrum = results[0]
```

If you want to run CE on large datasets, you can do

```python
proc.large_run(data, max_memory=1e9)
```

instead of `run`, which will ensure that the memory limit (1 GB in this case) is not exceeded on the GPU (unless of course you have other processes running).

2.4 Lomb-Scargle periodogram

The Lomb-Scargle periodogram ([Barning1963], [Vanicek1969], [Scargle1982], [Lomb1976]) is one of the best known and most popular period finding algorithms used in astrononomy. If you would like to learn more about least-squares methods for periodic signals, see the review article by [VanderPlas2017].

The LS periodogram is a least-squares estimator for the following model

$$\hat{y}(t|\omega, \theta) = \theta_1 \cos \omega t + \theta_2 \sin \omega t$$

and it is equivalent to the Discrete Fourier Transform in the regularly-sampled limit. For irregularly sampled data, LS is a maximum likelihood estimator for the parameters θ in the case where the noise is Gaussian. The periodogram has many normalizations in the literature, but `cuvarbase` adopts

$$P(\omega) = \frac{\chi_0^2 - \chi^2(\omega)}{\chi_0^2}$$

where

$$\chi^2(\omega) = \sum_i \left(\frac{y_i - \hat{y}(t_i|\omega, \theta)}{\sigma_i} \right)^2$$

is the goodness-of-fit statistic for the optimal parameters θ and

$$\chi_0^2 = \sum_i \left(\frac{y_i - \bar{y}}{\sigma_i}\right)^2$$

is the goodness-of-fit statistic for a constant fit, and $\bar{y}$ is the weighted mean,

$$\bar{y} = \sum_i w_i y_i$$

where $w_i \propto 1/\sigma_i^2$ and $\sum_i w_i = 1$.

The closed form of the periodogram is given by

$$P(\omega) = \frac{1}{\chi_0^2}\left(\frac{YC_\tau^2}{CC_\tau} + \frac{YS_\tau^2}{SS_\tau}\right)$$

Where

$$YC_\tau = \sum_i w_i y_i \cos\omega(t_i - \tau)$$

$$YS_\tau = \sum_i w_i y_i \sin\omega(t_i - \tau)$$

$$CC_\tau = \sum_i w_i \cos^2\omega(t_i - \tau)$$

$$SS_\tau = \sum_i w_i \sin^2\omega(t_i - \tau)$$

$$\tan 2\omega\tau = \frac{\sum_i w_i \sin 2\omega t_i}{\sum_i w_i \sin 2\omega t_i}$$

For the original formulation of the Lomb-Scargle periodogram without the constant offset term.

2.4.1 Adding a constant offset

Lomb-Scargle can be extended in many ways, most commonly to include a constant offset [ZK2009].

$$\hat{y}^{\mathrm{GLS}}(t|\omega,\theta) = \theta_1 \cos\omega t + \theta_2 \sin\omega t + \theta_3$$

This protects against cases where the mean of the data does not correspond with the mean of the underlying signal, as is usually the case with sparsely sampled data or for signals with large amplitudes that become too bright or dim to be observed during part of the signal phase.

With the constant offset term, the closed-form solution to $P(\omega)$ is the same, but the terms are slightly different. Derivations of this are in [ZK2009].

2.4.2 Getting $\mathcal{O}(N\log N)$ performance

The secret to Lomb-Scargle's speed lies in the fact that computing it requires evaluating sums that, for regularly-spaced data, can be evaluated with the fast Fourier transform (FFT), which scales as $\mathcal{O}(N_f \log N_f)$ where N_f is the number of frequencies. For *irregularly* spaced data, however, we can employ tricks to get to this scaling.

1. We can "extirpolate" the data with Legendre polynomials to a regular grid and then perform the FFT [PressRybicki1989], or,
2. We can use the non-equispaced fast Fourier transform (NFFT) [DuttRokhlin1993], which is tailor made for this exact problem.

The latter was shown by [Leroy2012] to give roughly an order-of-magnitude speed improvement over the [PressRybicki1989] method, with the added benefit that the NFFT is a rigorous extension of the FFT and has proven error bounds.

It's worth mentioning the [Townsend2010] CUDA implementation of Lomb-Scargle, however this uses the $\mathcal{O}(N_{\rm obs}N_f)$ "naive" implementation of LS without any FFT's.

2.4.3 Estimating significance

See [Baluev2008] for more information (TODO.)

2.4.4 Example: Basic

```python
import skcuda.fft
import cuvarbase.lombscargle as gls
import numpy as np
import matplotlib.pyplot as plt


t = np.sort(np.random.rand(300))
y = 1 + np.cos(2 * np.pi * 100 * t - 0.1)
dy = 0.1 * np.ones_like(y)
y += dy * np.random.randn(len(t))

# Set up LombScargleAsyncProcess (compilation, etc.)
proc = gls.LombScargleAsyncProcess()

# Run on single lightcurve
result = proc.run([(t, y, dy)])

# Synchronize all cuda streams
proc.finish()

# Read result!
freqs, ls_power = result[0]

############
# Plotting #
############

f, ax = plt.subplots()
ax.set_xscale('log')

ax.plot(freqs, ls_power)
ax.set_xlabel('Frequency')
ax.set_ylabel('Lomb-Scargle')
plt.show()
```

2.4.5 Example: Batches of lightcurves

```
import skcuda.fft
import cuvarbase.lombscargle as gls
import numpy as np
import matplotlib.pyplot as plt

nlcs = 9

def lightcurve(freq=100, ndata=300):
        t = np.sort(np.random.rand(ndata))
        y = 1 + np.cos(2 * np.pi * freq * t - 0.1)
        dy = 0.1 * np.ones_like(y)
        y += dy * np.random.randn(len(t))
        return t, y, dy

freqs = 200 * np.random.rand(nlcs)
data = [lightcurve(freq=freq) for freq in freqs]

# Set up LombScargleAsyncProcess (compilation, etc.)
proc = gls.LombScargleAsyncProcess()

# Run on batch of lightcurves
results = proc.batched_run_const_nfreq(data)

# Synchronize all cuda streams
proc.finish()

############
# Plotting #
############
max_n_cols = 4
ncols = max([1, min([int(np.sqrt(nlcs)), max_n_cols])])
nrows = int(np.ceil(float(nlcs) / ncols))
f, axes = plt.subplots(nrows, ncols,
                       figsize=(3 * ncols, 3 * nrows))

for (frqs, ls_power), ax, freq in zip(results,
                                      np.ravel(axes),
                                      freqs):
        ax.set_xscale('log')
        ax.plot(frqs, ls_power)
        ax.axvline(freq, ls=':', color='r')

f.text(0.05, 0.5, "Lomb-Scargle", rotation=90,
       va='center', ha='right', fontsize=20)
f.text(0.5, 0.05, "Frequency",
       va='top', ha='center', fontsize=20)


for i, ax in enumerate(np.ravel(axes)):
        if i >= nlcs:
                ax.axis('off')
f.tight_layout()
f.subplots_adjust(left=0.1, bottom=0.1)
plt.show()
```

2.5 Box least squares (BLS) periodogram

The box-least squares periodogram [BLS] searches for the periodic dips in brightness that occur when, e.g., a planet passes in front of its host star. The algorithm fits a boxcar function to the data. The parameters used are

- `q`: the transit duration as a fraction of the period $t_{\rm trans}/P$
- `phi0`: the phase offset of the transit (from 0)
- `delta`: the difference between the out-of-transit brightness and the brightness during transit
- `y0`: The out-of-transit brightness

2.5.1 Using `cuvarbase` BLS

```python
import cuvarbase.bls as bls
import numpy as np
import matplotlib.pyplot as plt


def phase(t, freq, phi0=0.):
    phi = (t * freq - phi0)
    phi -= np.floor(phi)

    return phi


def transit_model(t, freq, y0=0.0, delta=1., q=0.01, phi0=0.5):
    phi = phase(t, freq, phi0=phi0)
    transit = phi < q

    y = y0 * np.ones_like(t)
    y[transit] -= delta
    return y


def data(ndata=100, baseline=1, freq=10, sigma=1., **kwargs):
    t = baseline * np.sort(np.random.rand(ndata))
    y = transit_model(t, freq, **kwargs)
    dy = sigma * np.ones_like(t)

    y += dy * np.random.randn(len(t))

    return t, y, dy


def plot_bls_model(ax, y0, delta, q, phi0, **kwargs):
    phi_plot = np.linspace(0, 1, 50./q)
    y_plot = transit_model(phi_plot, 1., y0=y0,
                           delta=delta, q=q, phi0=phi0)

    ax.plot(phi_plot, y_plot, **kwargs)


def plot_bls_sol(ax, t, y, dy, freq, q, phi0, **kwargs):
    w = np.power(dy, -2)
    w /= sum(w)
```

(continues on next page)

(continued from previous page)

```
    phi = phase(t, freq, phi0=phi0)
    transit = phi < q

    def ybar(mask):
        return np.dot(w[mask], y[mask]) / sum(w[mask])

    y0 = ybar(~transit)
    delta = y0 - ybar(transit)

    ax.scatter((phi[~transit] + phi0) % 1.0, y[~transit],
               c='k', s=1, alpha=0.5)
    ax.scatter((phi[transit] + phi0) % 1.0, y[transit],
               c='r', s=1, alpha=0.5)
    plot_bls_model(ax, y0, delta, q, phi0, **kwargs)

    ax.set_xlim(0, 1)
    ax.set_xlabel('$\phi$ ($f = %.3f$)' % (freq))
    ax.set_ylabel('$y$')

# set the transit parameters
transit_kwargs = dict(freq=0.1,
                      q=0.1,
                      y0=10.,
                      sigma=0.002,
                      delta=0.05,
                      phi0=0.5)

# generate data with a transit
t, y, dy = data(ndata=300,
                baseline=365.,
                **transit_kwargs)

# set up search parameters
search_params = dict(qmin=1e-2,
                     qmax=0.5,

                     # The logarithmic spacing of q
                     dlogq=0.1,

                     # Number of overlapping phase bins
                     # to use for finding the best phi0
                     noverlap=3)

# derive baseline from the data for consistency
baseline = max(t) - min(t)

# df ~ qmin / baseline
df = search_params['qmin'] / baseline
fmin = 2. / baseline
fmax = 2.

nf = int(np.ceil((fmax - fmin) / df))
freqs = fmin + df * np.arange(nf)

bls_power, sols = bls.eebls_gpu(t, y, dy, freqs,
                                **search_params)
```

(continues on next page)

(continued from previous page)

```
# best BLS fit
q_best, phi0_best = sols[np.argmax(bls_power)]
f_best = freqs[np.argmax(bls_power)]

# Plot results
f, (ax_bls, ax_true, ax_best) = plt.subplots(1, 3, figsize=(9, 3))

# Periodogram
ax_bls.plot(freqs, bls_power)
ax_bls.axvline(transit_kwargs['freq'],
               ls=':', color='k', label="$f_0$")
ax_bls.axvline(f_best, ls=':', color='r',
               label='BLS $f_{\\rm best}$')
ax_bls.set_xlabel('freq.')
ax_bls.set_ylabel('BLS power')

# True solution
plot_bls_sol(ax_true, t, y, dy,
             transit_kwargs['freq'],
             transit_kwargs['q'],
             transit_kwargs['phi0'])

# Best-fit solution
plot_bls_sol(ax_best, t, y, dy,
             f_best, q_best, phi0_best)


ax_true.set_title("True parameters")
ax_best.set_title("Best BLS parameters")

f.tight_layout()
plt.show()
```

2.5.2 A shortcut: assuming orbital mechanics

If you assume $R_p \ll R_\star$, $M_p \ll M_\star$, $L_p \ll L_\star$, and $e \ll 1$, where e is the ellipticity of the planetary orbit, L is the luminosity, R is the radius, and M mass, you can eliminate a free parameter.

This is because the orbital period obeys Kepler's third law,

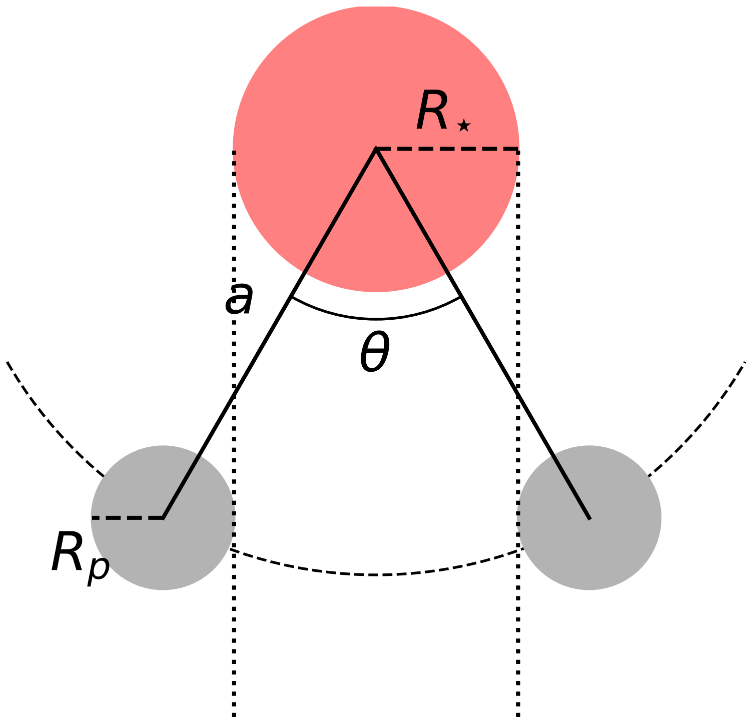
$$P^2 \approx \frac{4\pi^2 a^3}{G(M_p + M_\star)}$$

The angle of the transit is

$$\theta = 2\text{arcsin}\left(\frac{R_p + R_\star}{a}\right)$$

and q is therefore $\theta/(2\pi)$. Thus we have a relation between q and the period P

$$\sin \pi q = (R_p + R_\star)\left(\frac{4\pi^2}{P^2 G(M_p + M_\star)}\right)^{1/3}$$


$R_\star$
a
θ
R_p

By incorporating the fact that

$$R_\star = \left(\frac{3}{4\pi\rho_\star}\right)^{1/3} M_\star^{1/3}$$

where $\rho_\star$ is the average stellar density of the host star, we can write

$$\sin \pi q = \frac{(1+r)}{(1+m)^{1/3}} \left(\frac{3\pi}{G\rho_\star}\right)^{1/3} P^{-2/3}$$

where $r = R_p/R_\star$ and $m = M_p/M_\star$. We can get rid of the constant factors and convert this to more intuitive units to obtain

$$\sin \pi q \approx 0.238(1 + r - \frac{m}{3} + \dots) \left(\frac{\rho_\star}{\rho_\odot}\right)^{-1/3} \left(\frac{P}{\text{day}}\right)^{-2/3}$$

where here we've expanded $(1+r)/(1+m)^{1/3}$ to first order in r and m.

2.5.3 Using the Keplerian assumption in `cuvarbase`

```python
import cuvarbase.bls as bls
import numpy as np
import matplotlib.pyplot as plt


def phase(t, freq, phi0=0.):
    phi = (t * freq - phi0)
    phi -= np.floor(phi)

    return phi


def transit_model(t, freq, y0=0.0, delta=1., q=0.01, phi0=0.5):
    phi = phase(t, freq, phi0=phi0)
    transit = phi < q

    y = y0 * np.ones_like(t)
    y[transit] -= delta
    return y


def data(ndata=100, baseline=1, freq=10, sigma=1., **kwargs):
    t = baseline * np.sort(np.random.rand(ndata))
    y = transit_model(t, freq, **kwargs)
    dy = sigma * np.ones_like(t)

    y += dy * np.random.randn(len(t))

    return t, y, dy


def plot_bls_model(ax, y0, delta, q, phi0, **kwargs):
    phi_plot = np.linspace(0, 1, 50./q)
    y_plot = transit_model(phi_plot, 1., y0=y0,
                           delta=delta, q=q, phi0=phi0)
```

(continues on next page)

(continued from previous page)

```python
    ax.plot(phi_plot, y_plot, **kwargs)


def plot_bls_sol(ax, t, y, dy, freq, q, phi0, **kwargs):
    w = np.power(dy, -2)
    w /= sum(w)

    phi = phase(t, freq, phi0=phi0)
    transit = phi < q

    def ybar(mask):
        return np.dot(w[mask], y[mask]) / sum(w[mask])

    y0 = ybar(~transit)
    delta = y0 - ybar(transit)

    ax.scatter((phi[~transit] + phi0) % 1.0, y[~transit],
               c='k', s=1, alpha=0.5)
    ax.scatter((phi[transit] + phi0) % 1.0, y[transit],
               c='r', s=1, alpha=0.5)
    plot_bls_model(ax, y0, delta, q, phi0, **kwargs)

    ax.set_xlim(0, 1)
    ax.set_xlabel('$\phi$ ($f = %.3f$)' % (freq))
    ax.set_ylabel('$y$')

# the mean density of the host star in solar units
# i.e. rho = rho_star / rho_sun
rho = 1.

# set the transit parameters
transit_kwargs = dict(freq=2.,
                      q=bls.q_transit(2., rho=rho),
                      y0=10.,
                      sigma=0.005,
                      delta=0.01,
                      phi0=0.5)

# generate data with a transit
t, y, dy = data(ndata=300,
                baseline=365.,
                **transit_kwargs)

# set up search parameters
search_params = dict(
                     # Searches q values in the range
                     # (q0 * qmin_fac, q0 * qmax_fac)
                     # where q0 = q0(f, rho) is the fiducial
                     # q value for Keplerian transit around
                     # star with mean density rho
                     qmin_fac=0.5,
                     qmax_fac=2.0,

                     # Assumed mean stellar density
                     rho=1.0,

                     # The min/max frequencies as a fraction
```

(continues on next page)

(continued from previous page)

```
                      # of their autoset values
                      fmin_fac=1.0,
                      fmax_fac=1.5,

                      # oversampling factor; frequency spacing
                      # is multiplied by 1/samples_per_peak
                      samples_per_peak=2,

                      # The logarithmic spacing of q
                      dlogq=0.1,

                      # Number of overlapping phase bins
                      # to use for finding the best phi0
                      noverlap=3)

# Run keplerian BLS; frequencies are automatically set!
freqs, bls_power, sols = bls.eebls_transit_gpu(t, y, dy,
                                               **search_params)

# best BLS fit
q_best, phi0_best = sols[np.argmax(bls_power)]
f_best = freqs[np.argmax(bls_power)]

# Plot results
f, (ax_bls, ax_true, ax_best) = plt.subplots(1, 3, figsize=(9, 3))

# Periodogram
ax_bls.plot(freqs, bls_power)
ax_bls.axvline(transit_kwargs['freq'],
               ls=':', color='k', label="$f_0$")
ax_bls.axvline(f_best, ls=':', color='r',
               label='BLS $f_{\\rm best}$')
ax_bls.set_xlabel('freq.')
ax_bls.set_ylabel('BLS power')
ax_bls.set_xscale('log')

# True solution
label_true = '$q=%.3f$, ' % (transit_kwargs['q'])
label_true += '$\\phi_0=%.3f$' % (transit_kwargs['phi0'])
plot_bls_sol(ax_true, t, y, dy,
             transit_kwargs['freq'],
             transit_kwargs['q'],
             transit_kwargs['phi0'],
             label=label_true)
ax_true.legend(loc='best')

label_best = '$q=%.3f$, ' % (q_best)
label_best += '$\\phi_0=%.3f$' % (phi0_best)
# Best-fit solution
plot_bls_sol(ax_best, t, y, dy,
             f_best, q_best, phi0_best,
             label=label_best)
ax_best.legend(loc='best')

ax_true.set_title("True parameters")
ax_best.set_title("Best BLS parameters")
```

(continues on next page)

(continued from previous page)

```
f.tight_layout()
plt.show()
```

2.5.4 Period spacing considerations

The frequency spacing δf needed to resolve a BLS signal with width q, is

$$\delta f \lesssim \frac{q}{T}$$

where T is the baseline of the observations ($T = \max(t) - \min(t)$). This can be especially problematic if no assumptions are made about the nature of the signal (e.g., a Keplerian assumption). If you want to resolve a transit signal with a few observations, the minimum q value that you would need to search is $\propto 1/N$ where N is the number of observations.

For a typical Lomb-Scargle periodogram, the frequency spacing is $\delta f \lesssim 1/T$, so running a BLS spectrum with an adequate frequency spacing over the same frequency range requires a factor of $\mathcal{O}(N)$ more trial frequencies, each of which requiring $\mathcal{O}(N)$ computations to estimate the best fit BLS parameters. That means that BLS scales as $\mathcal{O}(N^2N_f)$ while Lomb-Scargle only scales as $\mathcal{O}(N_f \log N_f)$

However, if you can use the assumption that the transit is caused by an edge-on transit of a circularly orbiting planet, we not only eliminate a degree of freedom, but (assuming $\sin \pi q \approx \pi q$)

$$\delta f \propto q \propto f^{2/3}$$

The minimum frequency you could hope to measure a transit period would be $f_{\min} \approx 2/T$, and the maximum frequency is determined by $\sin \pi q < 1$ which implies

$$f_{max} = 8.612 \text{ c/day } \times \left(1 - \frac{3r}{2} + \frac{m}{2} - \dots\right) \sqrt{\frac{\rho_\star}{\rho_\odot}}$$

For a 10 year baseline, this translates to 2.7×10^5 trial frequencies. The number of trial frequencies needed to perform Lomb-Scargle over this frequency range is only about 3.1×10^4, so 8-10 times less. However, if we were to search the *entire* range of possible q values at each trial frequency instead of making a Keplerian assumption, we would instead require 5.35×10^8 trial frequencies, so the Keplerian assumption reduces the number of frequencies by over 1,000.

2.6 API documentation

2.6.1 cuvarbase package

Subpackages

cuvarbase.tests package

Submodules

cuvarbase.tests.test_bls module

cuvarbase.tests.test_ce module

cuvarbase.tests.test_lombscargle module

cuvarbase.tests.test_nfft module

cuvarbase.tests.test_pdm module

Module contents

Submodules

cuvarbase.bls module

cuvarbase.ce module

cuvarbase.core module

cuvarbase.cunfft module

cuvarbase.lombscargle module

cuvarbase.pdm module

cuvarbase.utils module

Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[G2013] Graham et al. 2013

[DuttRokhlin1993] Dutt, A., & Rokhlin, V. 1993, SIAM J. Sci. Comput., 14(6), 1368–1393.

[PressRybicki1989] Press, W. H., & Rybicki, G. B. 1989, ApJ, 338, 277

[Baluev2008] Baluev, R. V. 2008, MNRAS, 385, 1279

[ZK2009] Zechmeister, M., & Kürster, M. 2009, AAP, 496, 577

[VanderPlas2017] VanderPlas, J. T. 2017, arXiv:1703.09824

[Leroy2012] Leroy, B. 2012, AAP, 545, A50

[Townsend2010] Townsend, R. H. D. 2010, ApJS, 191, 247

[Barning1963] Barning, F. J. M. 1963, BAN, 17, 22

[Vanicek1969] Vaníček, P. 1969, APSS, 4, 387

[Scargle1982] Scargle, J. D. 1982, ApJ, 263, 835

[Lomb1976] Lomb, N. R. 1976, APSS, 39, 447

[BLS] Kovacs et al. 2002