BIASD Documentation

Release 0.1.1

Colin Kinz-Thompson

Nov 09, 2017

Contents

1 BIASD

BIASD allows you to analyze Markovian signal versus time series, such as those collected in single-molecule biophysics experiments, even when the kinetics of the underlying Markov chain are faster than the signal acquisition rate. The code here has been written in python for easy implementation, but unfortunately, the likelihood function is computationally expensive since it involves a numerical integral. Therefore, the likelihood function is also provided as C code and also in CUDA with python wrappers to use them with the rest of the code base.

2 Contents:

2.1 Getting Started

In general, BIASD uses the SMD data format (DOI: 10.1186/s12859-014-0429-4) for data storage, though this is not required. It also uses *emcee* (arXiv:1202.3665) to perform the Markov chain Monte Carlo (MCMC), though the Laplace approximation is also provided, which does not use emcee. To start using BIASD, you will need to first install some Python packages, and maybe compile some libraries if your want reasonable computational performance.

Software Requirements

Most of BIASD is written in Python. In general, you can run a Python script (.py) from a terminal with

```
python myscript.py
```

BIASD depends upon several Python packages, which must be installed. These can easily be obtained with the *conda* package manager, which is most efficiently obtained by installing Continuum Analytics' Miniconda. After installing *Miniconda* (the 64 bit, Python 2.7 version), you can then get the Python packages you will need.

You will need to use *conda* to install *pip* (version 9.0), *numpy* (version 1.11), *scipy* (verison 0.18), and *matplotlib* (version 1.5). For saving in the HDF5 SMD format, you will also need *h5py* (version 2.6) – these versions are up-to-date as of writing. In a terminal, install these packages with

```
conda install pip numpy scipy matplotlib h5py
```

Once you have *pip* installed, you can install *emcee* for MCMC and *corner* for plotting purposes from a terminal window with

```
pip install emcee corner
```

Adding BIASD to the Python Path

In order for your version of Python to find BIASD, you'll need to add the folder containing the BIASD code into your environmental variable PYTHONPATH. To do this, you'll need to edit your shell startup script with a text editor. This is located at */Users/<username>/.profile* for mac, or */home/<username>/.bashrc* if you're using bash for linux/unix; here, you should replace *<username>* with your username. At the end of your file, add the line

```
export PYTHONPATH="${PYTHONPATH}:/path/to/the/biasd/folder/you/downloaded"
```

which assumes that you downloaded the BIASD package to */path/to/the/biasd/folder/you/downloaded*. Note, that you need to either restart your terminal window, or source the startup script, e.g.

```
source ~/.bashrc
```

After this, you should be able to use BIASD with Python. However, you should compile the likelihood function if you want better performance.

2.2 Compiling the Likelihood

Background

The BIASD log-likelihood function is something like

$$ln(\mathcal{L}) \sim \sum_t ln\left(\delta(f) + \delta(1-f) + \int_0^1 df \cdot \text{blurring}\right)$$

Unfortunately, the integral in the logarithm makes it difficult to compute. It is the rate limiting step for this calculation, which is quite slow in Python. Therefore, this package comes with the log-likelihood function written in C, and also in CUDA. There are three versions in the `./biasd/src` directory. One is in pure C – it should be fairly straight forward to compile. The second is written in C with the GNU Science Library (GSL) – it's slightly faster, but requires having installed GSL. The third is in CUDA, which allows the calculations to be performed on NVIDIA GPUs. You can use any of the above if compiled, or a version written in Python if you don't want to compile anything.

How to Compile

There's a Makefile included in the package that will allow you to easily compile all of the libraries necessary to calculate BIASD likelihoods. First, to download GSL, go to their FTP site and download the latest version. Un-pack it, then in the terminal, navigate to the directory using `cd` and type

```
./configure
make
make install
```

Now, even if you didn't install GSL, you can compile the BIASD likelihood functions. In the terminal, move to the BIASD directory using `cd`, and make them with

```
make
```

Some might fail, for instance if you don't have a CUDA-enabled GPU, but you'll compile as many as possible into the `./biasd/lib` directory.

Testing Speed

To get a feeling for how long it takes the various versions of the BIASD likelihood function to execute, you can use the test function in the likelihood module. For instance, try

```
import biasd as b

# Switch to the Python version
b.likelihood.use_python_ll()

# Run the test 10 times, for 5000 datapoints
b.likelihood.test_speed(10,5000)

# Switch to the C version and test
# Note: will default to GSL over pure C
b.likelihood.use_C_ll()
b.likelihood.test_speed(10,5000)

# Switch to the CUDA version and test
```

```
b.likelihood.use_CUDA_ll()
b.likelihood.test_speed(10,5000)
```

The actual execution time depends upon the rate constants, but Python is ~ 1 ms, C with GSL is around ~ 50 us, and CUDA (when you have many datapoints) is ~ 1 us.

2.3 BIASD Examples

Here are some example Python scripts to perform BIASD. They can be found in *./example_data*, along with simulated data in a tab-delimited format (*./example_data/example_data.dat*), and an example HDF5 SMD dataset containing this data and some analysis results (*./example_data/example_dataset.hdf5*).

Creating a new SMD file

This script loads the example data, and then creates an HDF5 SMD data file to contain this data. Future analysis performed with BIASD can also be saved into this file.

```
## Imports
import numpy as np
import biasd as b

## Create a new SMD file
filename = './example_dataset.hdf5'
dataset = b.smd.new(filename)

## Load example trajectories (N,T)
example_data = b.smd.loadtxt('example_data.dat')
n_molecules, n_datapoints = example_data.shape

## These signal versus time trajectories were simulated to be like smFRET data.
## The simulation parameters were:
tau = 0.1 # seconds
e1 = 0.1 # E_{FRET}
e2 = 0.9 # E_{FRET}
sigma = 0.05 #E_{FRET}
k1 = 3. # s^{-1}
k2 = 8. # s^{-1}

truth = np.array((e1,e2,sigma,k1,k2))

## Create a vector with the time of each datapoint
time = np.arange(n_datapoints) * tau

## Add the trajectories to the SMD file automatically
b.smd.add.trajectories(dataset, time, example_data, x_label='time', y_label='E_{FRET}
↪')

## Add some metadata about the simulation to each trajectory
for i in range(n_molecules):

        # Select the group of interest
        trajectory = dataset['trajectory ' + str(i)]

        # Add an attribute called tau to the data group.
        # This group contains the time and signal vectors.
```

```
        trajectory['data'].attrs['tau'] = tau

        # Add a new group called simulation in the data group
        simulation = trajectory['data'].create_group('simulation')

        # Add relevant simulation paramters
        simulation.attrs['tau'] = tau
        simulation.attrs['e1'] = e1
        simulation.attrs['e2'] = e2
        simulation.attrs['sigma'] = sigma
        simulation.attrs['k1'] = k1
        simulation.attrs['k2'] = k2

        # Add an array of simulation parameters for easy access
        simulation.attrs['truth'] = truth

## Save the changes, and close the HDF5 file
b.smd.save(dataset)
```

Sample the posterior with MCMC

This script loads the example data from above, sets some priors, and then uses the Markov chain Monte Carlo (MCMC) technique to sample the posterior.

```
## Imports
import matplotlib.pyplot as plt
import numpy as np
import biasd as b


#### Setup the analysis
## Load the SMD example dataset
filename = './example_dataset.hdf5'
dataset = b.smd.load(filename)

## Select the data from the first trajectory
trace = dataset['trajectory 0']
time = trace['data/time'].value
fret = trace['data/E_{FRET}'].value

## Parse meta-data to load time resolution
tau = trace['data'].attrs['tau']

## Get the simulation ground truth values
truth = trace['data/simulation'].attrs['truth']

## Close the dataset
dataset.close()


#### Perform a Calculation
## Make the prior distribution
## set means to ground truths: (.1, .9, .05, 3., 8.)
e1 = b.distributions.normal(0.1, 0.2)
e2 = b.distributions.normal(0.9, 0.2)
sigma = b.distributions.gamma(1., 1./0.05)
```

```
k1 = b.distributions.gamma(1., 1./3.)
k2 = b.distributions.gamma(1., 1./8.)
priors = b.distributions.parameter_collection(e1,e2,sigma,k1,k2)

## Setup the MCMC sampler to use 100 walkers and 4 CPUs
nwalkers = 100
ncpus = 4
sampler, initial_positions = b.mcmc.setup(fret, priors, tau, nwalkers, threads =␣
↪ncpus)

## Burn-in 100 steps and then remove them form the sampler,
## but keep the final positions
sampler, burned_positions = b.mcmc.burn_in(sampler,initial_positions,nsteps=100)

## Run 100 steps starting at the burned-in positions. Timing data will provide an␣
↪idea of how long each step takes
sampler = b.mcmc.run(sampler,burned_positions,nsteps=100,timer=True)

## Continue on from step 100 for another 900 steps. Don't display timing.
sampler = b.mcmc.continue_run(sampler,900,timer=False)

## Get uncorrelated samples from the chain by skipping samples according to the␣
↪autocorrelation time of the variable with the largest autocorrelation time
uncorrelated_samples = b.mcmc.get_samples(sampler,uncorrelated=True)

## Make a corner plot of these uncorrelated samples
fig = b.mcmc.plot_corner(uncorrelated_samples)
fig.savefig('example_mcmc_corner.png')


#### Save the analysis
## Create a new group to hold the analysis in 'trajectory 0'
dataset = b.smd.load(filename)
trace = dataset['trajectory 0']
mcmc_analysis = trace.create_group("MCMC analysis 20170106")

## Add the priors
b.smd.add.parameter_collection(mcmc_analysis,priors,label='priors')

## Extract the relevant information from the sampler, and save this in the SMD file.
result = b.mcmc.mcmc_result(sampler)
b.smd.add.mcmc(mcmc_analysis,result,label='MCMC posterior samples')

## Save and close the dataset
b.smd.save(dataset)
```

Laplace approximation and computing the predictive posterior

This script loads the example data, sets some priors, and then finds the Laplace approximation to the posterior distribution. After this, it uses samples from this posterior to calculate the predictive posterior, which is the probability distribution for where you would expect to find new data.

```
## Imports
import matplotlib.pyplot as plt
import numpy as np
import biasd as b
```

```
#### Setup the analysis
## Load the SMD example dataset
filename = './example_dataset.hdf5'
dataset = b.smd.load(filename)

## Select the data from the first trajectory
trace = dataset['trajectory 0']
time = trace['data/time'].value
fret = trace['data/E_{FRET}'].value

## Parse meta-data to load time resolution
tau = trace['data'].attrs['tau']

## Get the simulation ground truth values
truth = trace['data/simulation'].attrs['truth']

## Close the dataset
dataset.close()


#### Perform a Calculation
## Make the prior distribution
## set means to ground truths: (.1, .9, .05, 3., 8.)
e1 = b.distributions.normal(0.1, 0.2)
e2 = b.distributions.normal(0.9, 0.2)
sigma = b.distributions.gamma(1., 1./0.05)
k1 = b.distributions.gamma(1., 1./3.)
k2 = b.distributions.gamma(1., 1./8.)
priors = b.distributions.parameter_collection(e1,e2,sigma,k1,k2)

## Find the Laplace approximation to the posterior
posterior = b.laplace.laplace_approximation(fret,priors,tau)

## Calculate the predictive posterior distribution for visualization
x = np.linspace(-.2,1.2,1000)
samples = posterior.samples(100)
predictive = b.likelihood.predictive_from_samples(x,samples,tau)


#### Save this analysis
## Load the dataset file
dataset = b.smd.load(filename)

## Create a new group to hold the analysis in 'trajectory 0'
trace = dataset['trajectory 0']
laplace_analysis = trace.create_group("Laplace analysis 20161230")

## Add the priors
b.smd.add.parameter_collection(laplace_analysis,priors,label='priors')

## Add the posterior
b.smd.add.laplace_posterior(laplace_analysis,posterior,label='posterior')

## Add the predictive
laplace_analysis.create_dataset('predictive x',data = x)
laplace_analysis.create_dataset('predictive y',data = predictive)
```

```
## Save and close the dataset
b.smd.save(dataset)


#### Visualize the results
## Plot a histogram of the data
plt.hist(fret, bins=71, range=(-.2,1.2), normed=True, histtype='stepfilled', alpha=.6,
↪ color='blue', label='Data')

## Plot the predictive posterior of the Laplace approximation solution
plt.plot(x, predictive, 'k', lw=2, label='Laplace')

## We know the data was simulated, so:
## plot the probability distribution used to simulate the data
plt.plot(x, np.exp(b.likelihood.nosum_log_likelihood(truth, x, tau)), 'r', lw=2,␣
↪label='Truth')

## Label Axes and Curves
plt.ylabel('Probability',fontsize=18)
plt.xlabel('Signal',fontsize=18)
plt.legend()

## Make the Axes Pretty
a = plt.gca()
a.spines['right'].set_visible(False)
a.spines['top'].set_visible(False)
a.yaxis.set_ticks_position('left')
a.xaxis.set_ticks_position('bottom')

# Save the figure, then show it
plt.savefig('example_laplace_predictive.png')
plt.show()
```

2.4 BIASD GUI

Also included is a graphical user interface to facilitate analysis with BIASD. Code for it is found in the *./biasd/gui* folder. It is written with PyQt5, which is a Python binding for Qt5. Thus, you'll need PyQt5 installed to run it, which you can get from a terminal with

```
conda install PyQt5
```

Once you have this installed, you can invoke the GUI from Python with

```
import biasd as b
b.gui.launch()
```

There is a script to do this in the main directory called *./launch_GUI.py*. You can launch it from that directory with

..code-block:: bash

    python launch_GUI.py

or

..code-block:: bash

    ./launch_GUI.py

Functionality

Functionality in the GUI is not as rich as in the Python module, however, you can easily explore HDF5 SMD files, which is not trivial from a Python script.

3 Code Documentation:

3.1 Distributions

This page gives the details about the code in biasd.distributions.

Some standard probability distributions

Convert between distributions

Distributions can be collected for priors or posteriors

Collections can be visualized

Finally, you can easily generate a few useful collections using

3.2 Laplace

This page gives the details about the code in biasd.laplace.

Laplace Approximation

In order to calculate the Laplace approximation to the posterior probability distribution, you must calculate the second derivative of the log-posterior function at the maximum a posteriori (MAP) estimate. This module contains code to calculate the finite difference Hessian, find the MAP estimate of the BIASD log-posterior using numerical maximization (Nelder-Mead), and apply this analysis to a time series. You should probably only need to use the *biasd.laplace.laplace_approximation()* function.

3.3 Likelihood

This page gives the details about the code in biasd.likelihood.

Switch the log-likelihood function

There are two main functions that you use for BIASD in *biasd.likelihood*. One to calculate the log-likelihood function, and the other to calculate the log-posterior function, which relies on the log-likelihood function. However, in truth, there are several different version of the log-likehood function that all accept the same arguments and return the results. There's one written in Python, (two) written in C, and one written in for CUDA. Assuming that they are compiled (i.e., C or CUDA), you can toggle between them to choose which version the log-likelihood function uses. In general, you'll want to use the C version if you have only a few data points (< 500), since it is fast and it allows you to use multiple processors when performing MCMC with emcee. If you have a lot of data points, you'll probably want to use the CUDA version, where each CUDA-core calculates the log-likelihood of a single data point. Anyway, you can toggle between the versions using

```
import biasd as b

# Switch to the slow, python implementation
b.likelihood.use_python_ll()

# Switch to the medium, parallelizable C version
b.likelihood.use_c_ll()

# Switch to the high-throughput CUDA version
b.likelihood.use_cuda_ll()
```

Finally, you can test the speed per datapoint of each of these version with

If you're ever confused about which version you're using, you can check the *biasd.likelihood.ll_version* variable.

Warning: Changing *biasd.likelihood.ll_version* will not switch which likelihood function is being used.

Inference-related functions

3.4 MCMC

This page gives the details about the code in biasd.mcmc.

Markov chain Monte Carlo

To sample the posterior probability distribution in BIASD, we'll use an affine invariant Markov chain Monte Carlo (MCMC) sampler. The implementation here uses emcee, which allow very efficient MCMC sampling. It is described in

> **Title** emcee: The MCMC Hammer
>
> **Authors** Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, and Jonathan Goodman
>
> **arXiv** http://arxiv.org/abs/1202.3665
>
> **DOI** 10.1086/670067

Which extends upon the paper

> **Title** Ensemble samplers with affine invariance
>
> **Authors** Jonathan Goodman, and Jonathan Weare
>
> **Citation** *Comm. Appl. Math. Comp. Sci.* **2010**, *5(1)*, 65-80.
>
> **DOI** 10.2140/camcos.2010.5.65>

Setup and Run MCMC

Example use:

```
import biasd as b

# Load data
data = b.smd.load('data.smd')
tau = 0.1

# Get a molecule and priors
```

```
d = data.data[0].values.FRET
priors = b.distributions.guess_priors(d,tau)

# Setup the sampler for this molecule
# Use 100 walkers, and 4 CPUs
sampler, initial_positions = b.mcmc.setup(dy, priors, tau, 100, initialize='rvs',␣
↪threads=4)

# Burn-in 100 steps and then remove them, but keep the final positions
sampler,burned_positions = b.mcmc.burn_in(sampler,initial_positions,nsteps=100)

# Run 100 steps starting at the burned-in positions.
sampler = b.mcmc.run(sampler,burned_positions,nsteps=100)
# Continue on from step 100 for another 900 steps. Don't display timing
sampler = b.mcmc.continue_run(sampler,900,timer=False)

# Save the sampler data
result = b.mcmc.mcmc_result(sampler)
data = b.smd.add.mcmc(result)
b.smd.save('data.smd',data)
```

Analyze MCMC samples

Note, for the corner plot, you must have corner. Anyway, continuing on from the previous example...

Example:

```
# ...

# Calculate auto-correlation times for each variable
largest_autocorrelation_time = b.mcmc.chain_statistics(sampler)

# Collect uncorrelated samples from the sampler
samples = b.mcmc.get_samples(sampler)

# Plot the joint and marginalized distributions from the samples using corner, and␣
↪then save the figure
f = b.mcmc.plot_corner(samples)
plt.savefig('mcmc_test.pdf')

# Create a collection of the marginalized posterior distributions
posterior = b.mcmc.create_posterior_collection(samples,priors)

# View that collection
b.distributions.viewer(posterior)
```

3.5 SMD

This page gives the details about the code in biasd.io.

The Single-Molecule Dataset (SMD) format is a standardized data format for use in time-resolved, single-molecule experiments (e.g, smFRET, force spectroscopy, etc.). It was published in collaboration between the Gonzalez and Herschlag labs (Greenfield, M *et al*. *BMC Bioinformatics* **2015**, *16*, 3. DOI: 10.1186/s12859-014-0429-4). Here we have adapted the SMD data format for use with the HDF5 data format, which we access with the python library h5py.

Work with SMD data

This module allows you to use the single-molecule dataset (SMD) format, and integrate BIASD results into them. The data is stored in as HDF5, and as such can be arbitrarily changed to fit your needs. Generally speaking, the structure is

SMD File (HDF5 File):

- **attrs (HDF5 Attributes)**
  - time created (Str)
  - SMD hash ID (Str)
  - number of trajectories (Int)
  - attribute 1 (denotes generic attribute) (Anything)
  - attribute 2 (denotes generic attribute) (Anything)
  - ...
- **trajectory 0 (HDF5 Group)**
  - **attrs (HDF5 Attributes)**
    - time created
    - SMD hash ID
    - attribute 1
    - attribute 2
    - ...
  - **data (HDF5 Group)**
    - **time (HDF5 Dataset)**
      - value (numpy ndarray of values)
    - **signal 1 (HDF5 Dataset)**
      - value (numpy ndarray of values)
    - ...
  - **BIASD analysis 1 (HDF5 Group), eg ensemble MCMC results**
    - **attrs (HDF5 Attributes)**
      - time created
      - SMD hash ID
      - attribute 1
      - attribute 2
      - ...
    - **analysis dataset 1 (HDF5 Dataset), eg MCMC samples**
      - ...
- **trajectory 1 (HDF5 Dataset)**
  - ...

Add BIASD results to an SMD object

Read BIASD results from an SMD object into a useful format

Examples

Create, add, and save example

```
import biasd as b
import numpy as np

# Load the data
cy3 = b.smd.loadtxt('my_cy3_data.dat')
cy5 = b.smd.loadtxt('my_cy5_data.dat')
fret = d1/(d1+d2)

# Create the time axis, assuming data is in (N,T) array
tau = 0.1
t = np.arange(fret.shape[1])*tau


# Make a new HDF5 SMD file
filename = '20161230_fret_experiment_1.hdf5'
f = b.smd.new(filename)

# Add the FRET trajectories
b.smd.add.trajectories(f, t, fret, x_label='Time (s)', y_label='E_{FRET}')
nmolecules  = f.attrs['number of trajectories']

# Save and close the output file
b.smd.save(f)

# Setup the priors
e1 = b.distributions.beta(1.,9.)
e2 = b.distributions.beta(9.,1.)
sigma = b.distributions.gamma(2.,2./.05)
k1 = b.distributions.gamma(20.,20./3.)
k2 = b.distributions.gamma(20.,20./8.)
priors = b.distributions.parameter_collection(e1,e2,sigma,k1,k2)

# Open the output file
f = b.smd.open(filename)

# Loop over all the molecules
for i in range(nmolecules):

        # Open the file again, because saving closes it
        f = b.smd.load(filename)

        # Get the ith trajectory reference
        trajectory = f['trajectory %d'%(i)]

        # Get the entire (i.e. [:] at the end) E_{FRET} data vector
        data = trajectory['data/E_{FRET}'][:]
        # Or equivalently...
        data = trajectory['data/E_{FRET}'].value

        # Create a new group for this particular analysis
```

```
        group = trajectory.create_group("20161231 Laplace Analysis")

        # Add the value of tau to the file for reference
        group.attrs['tau'] = tau

        # Add the priors used to the file for reference
        b.smd.add.parameter_collection(group,priors,label="Priors")

        # Do some calculation on data... for instance,
        laplace_result = b.laplace.laplace_approximation(data,priors,tau)

        # Add the results to the file
        b.smd.add.laplace_posterior(group,laplace_result)

        # Save the results as we go in case of a crash
        b.smd.save(f)
```

Load, and read from example above

```
import biasd as b

# Load the SMD
filename ='20161230_fret_experiment_1.hdf5'
f = b.smd.load(filename)

# Read the priors for the first molecule (0)
analysis = f['trajectory 0/20161231 Laplace Analysis']
priors = b.smd.read.parameter_collection(analysis['Priors'])

# Read the Laplace posterior object for the first molecule (0)
lp = b.smd.read.laplace_posterior(analysis['result'])

# Or if there are also MCMC results called 'result' in a group called '20170101 MCMC␣
↪Analysis' in trajectory 1...
analysis = f['trajectory 1/20170101 MCMC Analysis']
mcmc_results = b.smd.read.mcmc(analysis['result'])
```

Python Module Index

S

Index

S