fgivenx Documentation

Release 2.2.1

Will Handley

Jun 18, 2020

Contents

fgivenx Functional Posterior Plotter

Author Will Handley

Version 2.2.1

Homepage https://github.com/williamjameshandley/fgivenx

Documentation http://fgivenx.readthedocs.io/

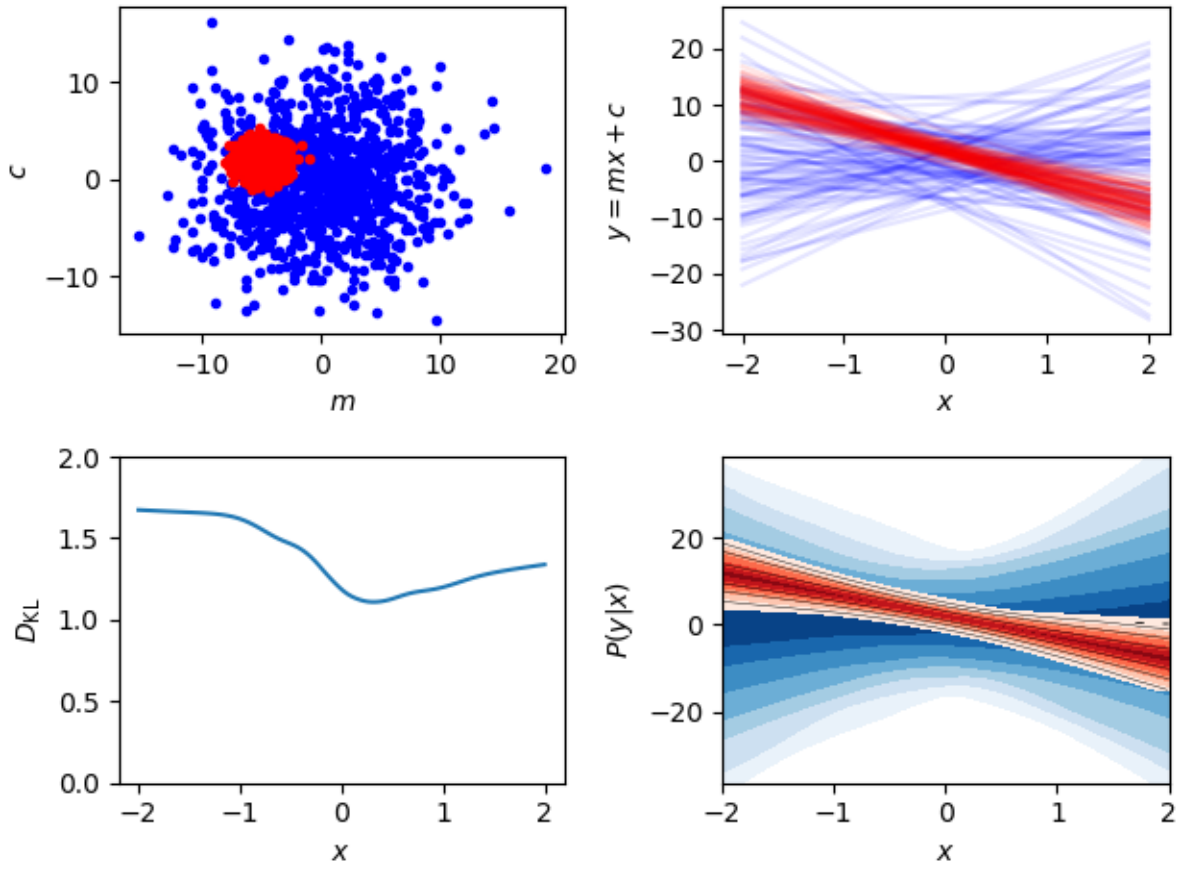
CHAPTER 1

Description

`fgivenx` is a python package for plotting posteriors of functions. It is currently used in astronomy, but will be of use to any scientists performing Bayesian analyses which have predictive posteriors that are functions.

This package allows one to plot a predictive posterior of a function, dependent on sampled parameters. We assume one has a Bayesian posterior `Post(theta|D,M)` described by a set of posterior samples `{theta_i}~Post`. If there is a function parameterised by theta `y=f(x;theta)`, then this script will produce a contour plot of the conditional posterior `P(y|x,D,M)` in the `(x,y)` plane.

The driving routines are `fgivenx.plot_contours`, `fgivenx.plot_lines` and `fgivenx.plot_dkl`. The code is compatible with getdist, and has a loading function provided by `fgivenx.samples_from_getdist_chains`.


c
m
y = mx + c
x
D_{KL}
x
P(y|x)
x

CHAPTER 2

Getting Started

Users can install using pip:

```
pip install fgivenx
```

from source:

```
git clone https://github.com/williamjameshandley/fgivenx
cd fgivenx
python setup.py install --user
```

or for those on Arch linux it is available on the AUR

You can check that things are working by running the test suite (You may encounter warnings if the optional dependency `joblib` is not installed):

```
pip install pytest pytest-runner pytest-mpl
export MPLBACKEND=Agg
pytest <fgivenx-install-location>

# or, equivalently
git clone https://github.com/williamjameshandley/fgivenx
cd fgivenx
python setup.py test
```

Check the dependencies listed in the next section are installed. You can then use the `fgivenx` module from your scripts.

Some users of OSX or Anaconda may find `QueueManagerThread` errors if Pillow is not installed (run `pip install pillow`).

If you want to use parallelisation, have progress bars or getdist compatibility you should install the additional optional dependencies:

```
pip install joblib tqdm getdist
# or, equivalently
pip install -r  requirements.txt
```

You may encounter warnings if you don't have the optional dependency `joblib` installed.

CHAPTER 3

Dependencies

Basic requirements:

- Python 2.7+ or 3.4+
- matplotlib
- numpy
- scipy

Documentation:

- sphinx
- numpydoc

Tests:

- pytest
- pytest-mpl

Optional extras:

- joblib (parallelisation) [+ pillow on some systems]
- tqdm (progress bars)
- getdist (reading of getdist compatible files)

CHAPTER 4

Documentation

Full Documentation is hosted at ReadTheDocs. To build your own local copy of the documentation you'll need to install sphinx. You can then run:

```
cd docs
make html
```

CHAPTER 5

Citation

If you use `fgivenx` to generate plots for a publication, please cite as:

```
Handley, (2018). fgivenx: A Python package for functional posterior
plotting . Journal of Open Source Software, 3(28), 849,
https://doi.org/10.21105/joss.00849
```

or using the BibTeX:

```
@article{fgivenx,
    doi = {10.21105/joss.00849},
    url = {http://dx.doi.org/10.21105/joss.00849},
    year  = {2018},
    month = {Aug},
    publisher = {The Open Journal},
    volume = {3},
    number = {28},
    author = {Will Handley},
    title = {fgivenx: Functional Posterior Plotter},
    journal = {The Journal of Open Source Software}
}
```

CHAPTER 6

Example Usage

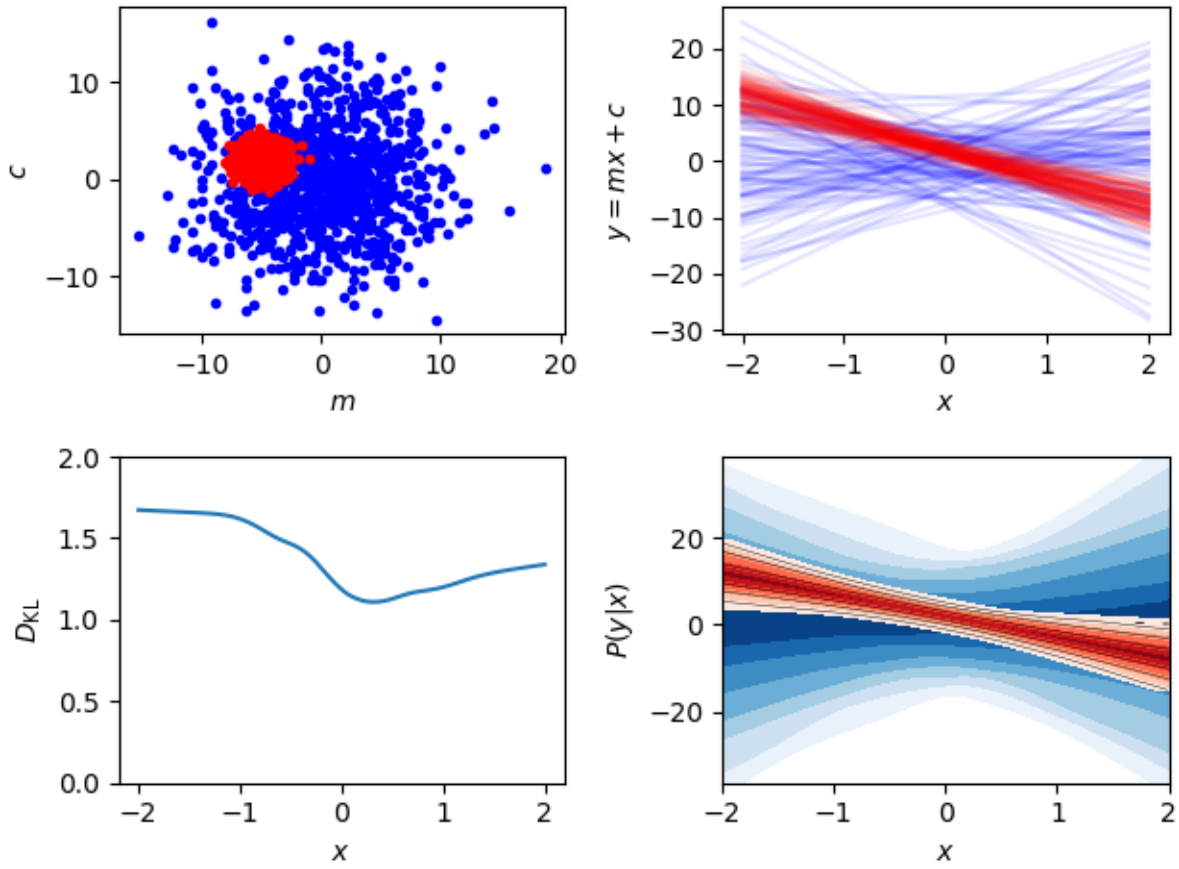
6.1 Plot user-generated samples

```
import numpy
import matplotlib.pyplot as plt
from fgivenx import plot_contours, plot_lines, plot_dkl


# Model definitions
# =================
# Define a simple straight line function, parameters theta=(m,c)
def f(x, theta):
    m, c = theta
    return m * x + c


numpy.random.seed(1)

# Posterior samples
nsamples = 1000
ms = numpy.random.normal(loc=-5, scale=1, size=nsamples)
cs = numpy.random.normal(loc=2, scale=1, size=nsamples)
samples = numpy.array([(m, c) for m, c in zip(ms, cs)]).copy()

# Prior samples
ms = numpy.random.normal(loc=0, scale=5, size=nsamples)
cs = numpy.random.normal(loc=0, scale=5, size=nsamples)
prior_samples = numpy.array([(m, c) for m, c in zip(ms, cs)]).copy()

# Set the x range to plot on
xmin, xmax = -2, 2
nx = 100
x = numpy.linspace(xmin, xmax, nx)
```

(continues on next page)

(continued from previous page)

```
# Set the cache
cache = 'cache/test'
prior_cache = cache + '_prior'

# Plotting
# ========
fig, axes = plt.subplots(2, 2)

# Sample plot
# -----------
ax_samples = axes[0, 0]
ax_samples.set_ylabel(r'$c$')
ax_samples.set_xlabel(r'$m$')
ax_samples.plot(prior_samples.T[0], prior_samples.T[1], 'b.')
ax_samples.plot(samples.T[0], samples.T[1], 'r.')

# Line plot
# ---------
ax_lines = axes[0, 1]
ax_lines.set_ylabel(r'$y = m x + c$')
ax_lines.set_xlabel(r'$x$')
plot_lines(f, x, prior_samples, ax_lines, color='b', cache=prior_cache)
plot_lines(f, x, samples, ax_lines, color='r', cache=cache)

# Predictive posterior plot
# -------------------------
ax_fgivenx = axes[1, 1]
ax_fgivenx.set_ylabel(r'$P(y|x)$')
ax_fgivenx.set_xlabel(r'$x$')
cbar = plot_contours(f, x, prior_samples, ax_fgivenx,
                     colors=plt.cm.Blues_r, lines=False,
                     cache=prior_cache)
cbar = plot_contours(f, x, samples, ax_fgivenx, cache=cache)

# DKL plot
# --------
ax_dkl = axes[1, 0]
ax_dkl.set_ylabel(r'$D_\mathrm{KL}$')
ax_dkl.set_xlabel(r'$x$')
ax_dkl.set_ylim(bottom=0, top=2.0)
plot_dkl(f, x, samples, prior_samples, ax_dkl,
         cache=cache, prior_cache=prior_cache)

ax_lines.get_shared_x_axes().join(ax_lines, ax_fgivenx, ax_samples)

fig.tight_layout()
fig.savefig('plot.png')
```



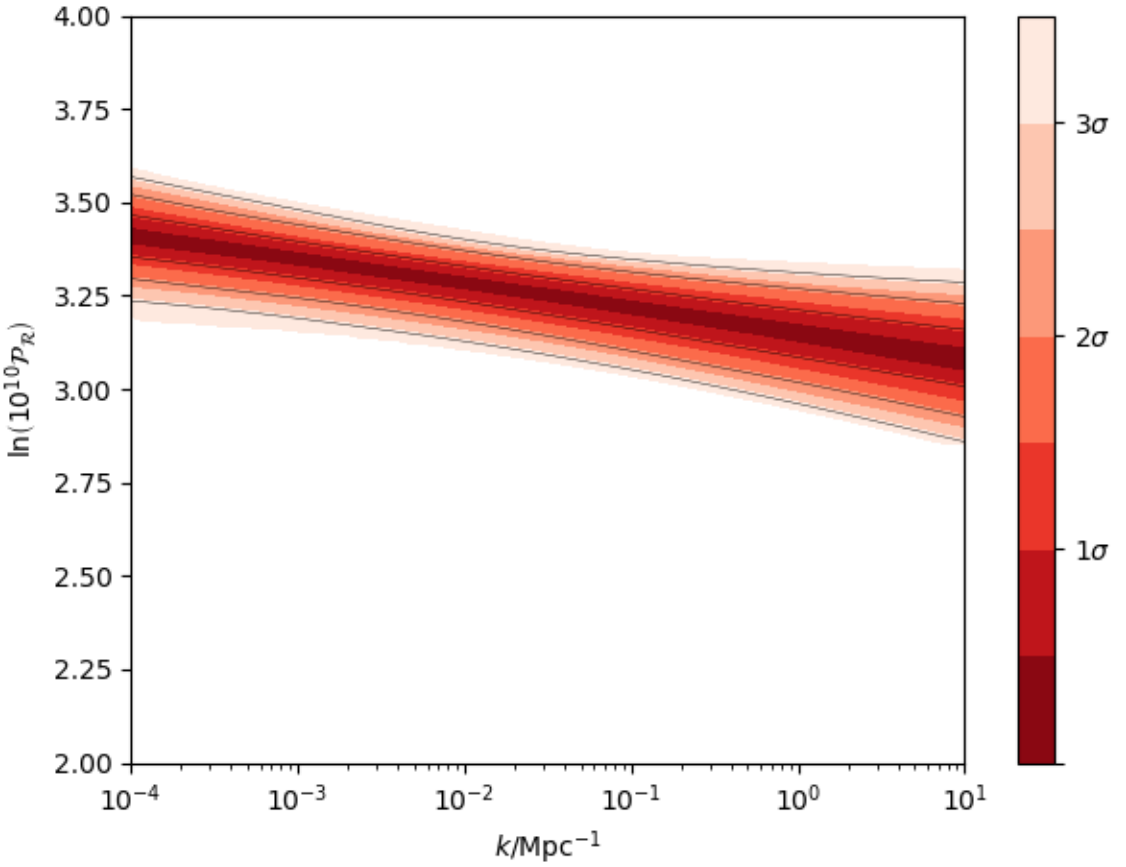
6.2 Plot GetDist chains

```
import numpy
import matplotlib.pyplot as plt
from fgivenx import plot_contours, samples_from_getdist_chains

file_root = './plik_HM_TT_lowl/base_plikHM_TT_lowl'
samples, weights = samples_from_getdist_chains(['logA', 'ns'], file_root)

def PPS(k, theta):
    logA, ns = theta
    return logA + (ns - 1) * numpy.log(k)

k = numpy.logspace(-4,1,100)
cbar = plot_contours(PPS, k, samples, weights=weights)
cbar = plt.colorbar(cbar,ticks=[0,1,2,3])
cbar.set_ticklabels(['',r'$1\sigma$',r'$2\sigma$',r'$3\sigma$'])

plt.xscale('log')
plt.ylim(2,4)
plt.ylabel(r'$\ln\left(10^{10}\mathcal{P}_\mathcal{R}\right)$')
plt.xlabel(r'$k / {\rm Mpc}^{-1}$')
plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.savefig('planck.png')
```



CHAPTER 7

Contributing

Want to contribute to `fgivenx`? Awesome! There are many ways you can contribute via the [GitHub repository](https://github.com/williamjameshandley/fgivenx), see below.

7.1 Opening issues

Open an issue to report bugs or to propose new features.

7.2 Proposing pull requests

Pull requests are very welcome. Note that if you are going to propose drastic changes, be sure to open an issue for discussion first, to make sure that your PR will be accepted before you spend effort coding it.

CHAPTER 8

Changelog

v2.2.0 Paper accepted

v2.1.17 100% coverage

v2.1.16 Tests fixes

v2.1.15 Additional plot tests

v2.1.13 Further bug fix in test suite for image comparison

v2.1.12 Bug fix in test suite for image comparison

v2.1.11 Documentation upgrades

v2.1.10 Added changelog

CHAPTER 9

fgivenx package

9.1 Module contents

The main driving routines for this package are:

- *plot_contours*
- *plot_lines*
- *plot_dkl*
- *samples_from_getdist_chains*

Example import and usage:

```
>>> import numpy
>>> from fgivenx import plot_contours, plot_lines, ...                    plot_dkl,␣
↪samples_from_getdist_chains
>>>
>>> file_root = '/my/getdist/file/root'
>>> params = ['m', 'c']
>>> samples = samples_from_getdist_chains(params, file_root)
>>> x = numpy.linspace(-1, 1, 100)
>>>
>>> def f(x, theta):
>>>     m, c = params
>>>     y = m * x + c
>>>     return y
>>>
>>> plot_contours(f, x, samples)
```

9.2 Submodules

9.3 fgivenx.drivers module

This module provides utilities for computing the grid for contours of a function reconstruction plot.

Required ingredients:

- sampled posterior probability distribution $P(\theta)$
- independent variable x
- dependent variable y
- functional form $y = f(x;\theta)$ parameterised by θ

Assuming that you have obtained samples of θ from an MCMC process, we aim to compute the density:

$$\begin{aligned} P(y|x) &= \int P(y = f(x;\theta)|x,\theta)P(\theta)d\theta \\ &= \int \delta(y - f(x;\theta))P(\theta)d\theta \end{aligned}$$

which gives our degree of knowledge for each $y = f(x;\theta)$ value given an x value.

In fact, for a more representative plot, we are not actually interested in the value of the probability density above, but in fact require the "iso-probablity posterior mass"

$$\mathrm{pmf}(y|x) = \int_{P(y'|x)<P(y|x)} P(y'|x)dy'$$

We thus need to compute this function on a rectangular grid of x and y.

fgivenx.drivers.**compute_dkl**(*f*, *x*, *samples*, *prior_samples*, ***kwargs*)

Compute the Kullback-Leibler divergence at each value of *x* for the prior and posterior defined by *prior_samples* and *samples*.

Parameters

f: function function $f(x;\theta)$ (or list of functions for each model) with dependent variable x, parameterised by θ.

x: 1D array-like x values to evaluate $f(x;\theta)$ at.

samples, prior_samples: 2D array-like θ samples (or list of θ samples) from posterior and prior to evaluate $f(x;\theta)$ at. *shape = (nsamples, npars)*

logZ: 1D array-like, optional log-evidences of each model if multiple models are passed. Should be same length as the list *f*, and need not be normalised. Default: *numpy.ones_like(f)*

weights, prior_weights: 1D array-like, optional sample weights (or list of weights), if desired. Should have length same as *samples.shape[0]*. Default: *numpy.ones_like(samples)*

ntrim: int, optional Approximate number of samples to trim down to, if desired. Useful if the posterior is dramatically oversampled. Default: None

cache, prior_cache: str, optional File roots for saving previous calculations for re-use

parallel, tqdm_args: see docstring for *fgivenx.parallel.parallel_apply()*

kwargs: further keyword arguments Any further keyword arguments are plotting keywords that are passed to *fgivenx.plot.plot()*.

Returns

1D numpy array: dkl values at each value of *x*.

fgivenx.drivers.**compute_pmf**(*f*, *x*, *samples*, ***kwargs*)

Compute the probability mass function given *x* at a range of *x* values for $y = f(x|\theta)$

$P(y|x) = \int P(y = f(x;\theta)|x,\theta)P(\theta)d\theta$

$\mathrm{pmf}(y|x) = \int_{P(y'|x)<P(y|x)} P(y'|x)dy'$

Additionally, if a list of log-evidences are passed, along with list of functions, samples and optional weights it marginalises over the models according to the evidences.

Parameters

f: function function $f(x;\theta)$ (or list of functions for each model) with dependent variable x, parameterised by θ.

x: 1D array-like *x* values to evaluate $f(x;\theta)$ at.

samples: 2D array-like θ samples (or list of θ samples) to evaluate $f(x;\theta)$ at. *shape = (nsamples, npars)*

logZ: 1D array-like, optional log-evidences of each model if multiple models are passed. Should be same length as the list *f*, and need not be normalised. Default: *numpy.ones_like(f)*

weights: 1D array-like, optional sample weights (or list of weights), if desired. Should have length same as *samples.shape[0]*. Default: *numpy.ones_like(samples)*

ny: int, optional Resolution of y axis. Default: *100*

y: array-like, optional Explicit descriptor of *y* values to evaluate. Default: *numpy.linspace(min(f), max(f), ny)*

ntrim: int, optional Approximate number of samples to trim down to, if desired. Useful if the posterior is dramatically oversampled. Default: None

cache: str, optional File root for saving previous calculations for re-use

parallel, tqdm_args: see docstring for *fgivenx.parallel.parallel_apply()*

Returns

1D numpy.array: *y* values pmf is computed at *shape=(len(y))* or *ny*

2D numpy.array: pmf values at each *x* and *y* *shape=(len(x),len(y))*

fgivenx.drivers.**compute_samples**(*f*, *x*, *samples*, ***kwargs*)

Apply the function(s) $f(x;\theta)$ to the arrays defined in *x* and *samples*. Has options for weighting, trimming, cacheing & parallelising.

Additionally, if a list of log-evidences are passed, along with list of functions, samples and optional weights it marginalises over the models according to the evidences.

Parameters

f: function function $f(x;\theta)$ (or list of functions for each model) with dependent variable x, parameterised by θ.

x: 1D array-like *x* values to evaluate $f(x;\theta)$ at.

samples: 2D array-like θ samples (or list of θ samples) to evaluate $f(x;\theta)$ at. *shape = (nsamples, npars)*

logZ: 1D array-like, optional log-evidences of each model if multiple models are passed. Should be same length as the list *f*, and need not be normalised. Default: *numpy.ones_like(f)*

weights: 1D array-like, optional sample weights (or list of weights), if desired. Should have length same as *samples.shape[0]*. Default: *numpy.ones_like(samples)*

ntrim: int, optional Approximate number of samples to trim down to, if desired. Useful if the posterior is dramatically oversampled. Default: None

cache: str, optional File root for saving previous calculations for re-use. Default: None

parallel, tqdm_args: see docstring for *fgivenx.parallel.parallel_apply()*

Returns

2D numpy.array Evaluate the function f at each x value and each theta. Equivalent to *[[f(x_i,theta) for theta in samples] for x_i in x]*

fgivenx.drivers.**plot_contours**(*f*, *x*, *samples*, *ax=None*, ***kwargs*)

Plot the probability mass function given x at a range of y values for $y = f(x|\theta)$

$P(y|x) = \int P(y = f(x;\theta)|x,\theta)P(\theta)d\theta$

$\mathrm{pmf}(y|x) = \int_{P(y'|x)<P(y|x)} P(y'|x)dy'$

Additionally, if a list of log-evidences are passed, along with list of functions, and list of samples, this function plots the probability mass function for all models marginalised according to the evidences.

Parameters

f: function function $f(x;\theta)$ (or list of functions for each model) with dependent variable x, parameterised by θ.

x: 1D array-like x values to evaluate $f(x;\theta)$ at.

samples: 2D array-like θ samples (or list of θ samples) to evaluate $f(x;\theta)$ at. *shape = (nsamples, npars)*

ax: axes object, optional `matplotlib.axes._subplots.AxesSubplot` to plot the contours onto. If unsupplied, then `matplotlib.pyplot.gca()` is used to get the last axis used, or create a new one.

logZ: 1D array-like, optional log-evidences of each model if multiple models are passed. Should be same length as the list *f*, and need not be normalised. Default: *numpy.ones_like(f)*

weights: 1D array-like, optional sample weights (or list of weights), if desired. Should have length same as *samples.shape[0]*. Default: *numpy.ones_like(samples)*

ny: int, optional Resolution of *y* axis. Default: *100*

y: array-like, optional Explicit descriptor of *y* values to evaluate. Default: *numpy.linspace(min(f), max(f), ny)*

ntrim: int, optional Approximate number of samples to trim down to, if desired. Useful if the posterior is dramatically oversampled. Default: None

cache: str, optional File root for saving previous calculations for re-use

parallel, tqdm_args: see docstring for *fgivenx.parallel.parallel_apply()*

kwargs: further keyword arguments Any further keyword arguments are plotting keywords that are passed to *fgivenx.plot.plot()*.

Returns

cbar: color bar `matplotlib.contour.QuadContourSet` Colors to create a global colour bar

fgivenx.drivers.**plot_dkl**(*f*, *x*, *samples*, *prior_samples*, *ax=None*, ***kwargs*)

Plot the Kullback-Leibler divergence at each value of x for the prior and posterior defined by *prior_samples* and *samples*.

Let the posterior be:

$P(y|x) = \int P(y = f(x;\theta)|x,\theta)P(\theta)d\theta$

and the prior be:

$Q(y|x) = \int P(y = f(x;\theta)|x,\theta)Q(\theta)d\theta$

then the Kullback-Leibler divergence at each x is defined by

$D_{\mathrm{KL}}(x) = \int P(y|x) \ln\left[\frac{Q(y|x)}{P(y|x)}\right] dy$

Additionally, if a list of log-evidences are passed, along with list of functions, and list of samples, this function plots the Kullback-Leibler divergence for all models marginalised according to the evidences.

Parameters

f: function function $f(x;\theta)$ (or list of functions for each model) with dependent variable x, parameterised by θ.

x: 1D array-like x values to evaluate $f(x;\theta)$ at.

samples, prior_samples: 2D array-like θ samples (or list of θ samples) from posterior and prior to evaluate $f(x;\theta)$ at. *shape = (nsamples, npars)*

ax: axes object, optional `matplotlib.axes._subplots.AxesSubplot` to plot the contours onto. If unsupplied, then `matplotlib.pyplot.gca()` is used to get the last axis used, or create a new one.

logZ: 1D array-like, optional log-evidences of each model if multiple models are passed. Should be same length as the list *f*, and need not be normalised. Default: *numpy.ones_like(f)*

weights, prior_weights: 1D array-like, optional sample weights (or list of weights), if desired. Should have length same as *samples.shape[0]*. Default: *numpy.ones_like(samples)*

ntrim: int, optional Approximate number of samples to trim down to, if desired. Useful if the posterior is dramatically oversampled. Default: None

cache, prior_cache: str, optional File roots for saving previous calculations for re-use

parallel, tqdm_args: see docstring for *fgivenx.parallel.parallel_apply()*

kwargs: further keyword arguments Any further keyword arguments are plotting keywords that are passed to *fgivenx.plot.plot()*.

fgivenx.drivers.**plot_lines**(*f*, *x*, *samples*, *ax=None*, ***kwargs*)

Plot a representative set of functions to sample

Additionally, if a list of log-evidences are passed, along with list of functions, and list of samples, this function plots the probability mass function for all models marginalised according to the evidences.

Parameters

f: function function $f(x;\theta)$ (or list of functions for each model) with dependent variable x, parameterised by θ.

x: 1D array-like x values to evaluate $f(x;\theta)$ at.

samples: 2D array-like θ samples (or list of θ samples) to evaluate $f(x;\theta)$ at. *shape = (nsamples, npars)*

ax: axes object, optional `matplotlib.axes._subplots.AxesSubplot` to plot the contours onto. If unsupplied, then `matplotlib.pyplot.gca()` is used to get the last axis used, or create a new one.

logZ: 1D array-like, optional log-evidences of each model if multiple models are passed. Should be same length as the list *f*, and need not be normalised. Default: *numpy.ones_like(f)*

weights: 1D array-like, optional sample weights (or list of weights), if desired. Should have length same as *samples.shape[0]*. Default: *numpy.ones_like(samples)*

ntrim: int, optional Approximate number of samples to trim down to, if desired. Useful if the posterior is dramatically oversampled. Default: None

cache: str, optional File root for saving previous calculations for re-use

parallel, tqdm_args: see docstring for *fgivenx.parallel.parallel_apply()*

kwargs: further keyword arguments Any further keyword arguments are plotting keywords that are passed to *fgivenx.plot.plot_lines()*.

9.4 fgivenx.dkl module

fgivenx.dkl.**DKL**(*arrays*)

Compute the Kullback-Leibler divergence from one distribution Q to another P, where Q and P are represented by a set of samples.

Parameters

arrays: tuple(1D numpy.array,1D numpy.array) samples defining distributions P & Q respectively

Returns

float: Kullback Leibler divergence.

fgivenx.dkl.**compute_dkl**(*fsamps*, *prior_fsamps*, ***kwargs*)

Compute the Kullback Leibler divergence for function samples for posterior and prior pre-calculated at a range of x values.

Parameters

fsamps: 2D numpy.array Posterior function samples, as computed by `fgivenx.compute_samples()`

prior_fsamps: 2D numpy.array Prior function samples, as computed by `fgivenx.compute_samples()`

parallel, tqdm_kwargs: optional see docstring for *fgivenx.parallel.parallel_apply()*.

cache: str, optional File root for saving previous calculations for re-use.

Returns

1D numpy.array: Kullback-Leibler divergences at each value of x. *shape=(len(fsamps))*

9.5 fgivenx.io module

class fgivenx.io.**Cache**(*file_root*)

Bases: `object`

Cacheing tool for saving recomputation.

Parameters

file_root: str cached values are saved in file_root.pkl

Methods

check(self, *args)	Check that the arguments haven't changed since the last call.
load(self)	Load cache from file using pickle.
save(self, *args)	Save cache to file using pickle.

check (*self*, **args*)
Check that the arguments haven't changed since the last call.

Parameters

***args:** All but the last argument are inputs to the cached function. The last is the actual value of the function.

Returns

If arguments unchanged: return the cached answer

else: indicate recomputation required by throwing a *CacheException*.

load (*self*)
Load cache from file using pickle.

save (*self*, **args*)
Save cache to file using pickle.

Parameters

***args:** All but the last argument are inputs to the cached function. The last is the actual value of the function.

exception fgivenx.io.**CacheChanged** (*file_root*)
Bases: *fgivenx.io.CacheException*

Exception to indicate the cache has changed.

exception fgivenx.io.**CacheException**
Bases: exceptions.Exception

Base exception to indicate cache errors

calling_function (*self*)
Get the name of the function calling this cache.

exception fgivenx.io.**CacheMissing** (*file_root*)
Bases: *fgivenx.io.CacheException*

Exception to indicate the cache does not exist.

exception fgivenx.io.**CacheOK** (*file_root*)
Bases: *fgivenx.io.CacheException*

Exception to indicate the cache can be used.

9.6 fgivenx.mass module

Utilities for computing the probability mass function.

fgivenx.mass.**PMF**(*samples*, *y*)

Compute the probability mass function.

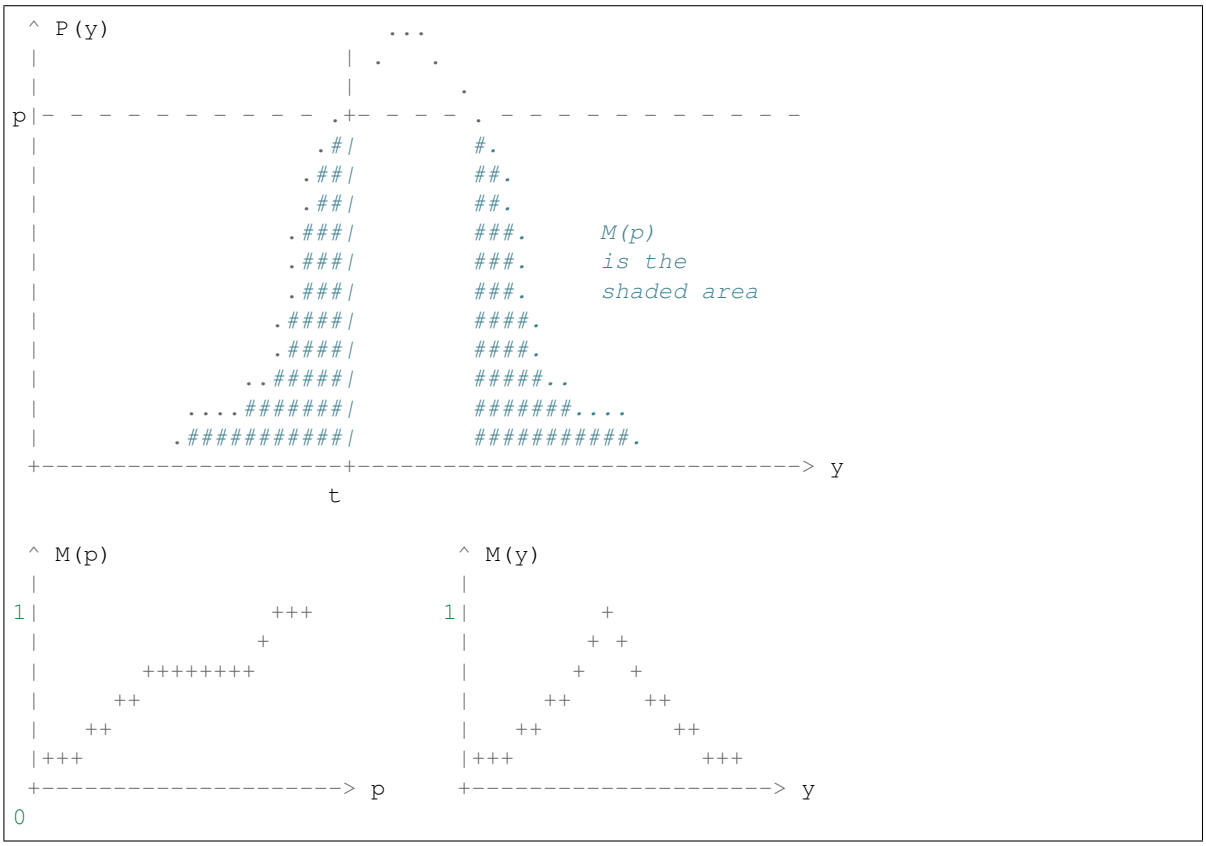
The set of samples defines a probability density P(y), which is computed using a kernel density estimator.

From $P(y)$ we define:

$\mathrm{pmf}(p) = \int_{P(y)<p} P(y)dy$

This is the cumulative distribution function expressed as a function of the probability

We aim to compute $M(y)$, which indicates the amount of probability contained outside the iso-probability contour passing through y:



Parameters

samples: array-like Array of samples from a probability density P(y).

y: array-like (optional) Array to evaluate the PDF at

Returns

1D numpy.array: PMF evaluated at each y value

fgivenx.mass.**compute_pmf**(*fsamps*, *y*, ***kwargs*)

Compute the pmf defined by fsamps at each x for each y.

Parameters

fsamps: 2D array-like array of function samples, as returned by `fgivenx.compute_samples()`

y: 1D array-like y values to evaluate the PMF at

parallel, tqdm_kwargs: optional see docstring for *fgivenx.parallel.parallel_apply()*.

Returns

2D numpy.array probability mass function at each x for each y *shape=(len(fsamps),len(y)*

9.7 fgivenx.parallel module

`fgivenx.parallel.`**`parallel_apply`**(*f*, *array*, ***kwargs*)

Apply a function to an array with openmp parallelisation.

Equivalent to *[f(x) for x in array]*, but parallelised if required.

Parameters

f: function Univariate function to apply to each element of array

array: array-like Array to apply f to

parallel: int or bool, optional int > 0: number of processes to parallelise over

int < 0 or bool=True: use OMP_NUM_THREADS to choose parallelisation

bool=False or int=0: do not parallelise

tqdm_kwargs: dict, optional additional kwargs for tqdm progress bars.

precurry: tuple, optional immutable arguments to pass to f before x, i.e. *[f(precurry,x) for x in array]*

postcurry: tuple, optional immutable arguments to pass to f after x i.e. *[f(x,postcurry) for x in array]*

Returns

list: *[f(precurry,x,postcurry) for x in array]* parallelised according to parallel

9.8 fgivenx.plot module

`fgivenx.plot.`**`plot`**(*x*, *y*, *z*, *ax=None*, ***kwargs*)

Plot iso-probability mass function, converted to sigmas.

Parameters

x, y, z [numpy arrays] Same as arguments to `matplotlib.pyplot.contour()`

ax: axes object, optional `matplotlib.axes._subplots.AxesSubplot` to plot the contours onto. If unsupplied, then `matplotlib.pyplot.gca()` is used to get the last axis used, or create a new one.

colors: color scheme, optional `matplotlib.colors.LinearSegmentedColormap` Color scheme to plot with. Recommend plotting in reverse (Default: `matplotlib.pyplot.cm.Reds_r`)

smooth: float, optional Percentage by which to smooth the contours. (Default: no smoothing)

contour_line_levels: List[float], optional Contour lines to be plotted. (Default: [1,2])

linewidths: float, optional Thickness of contour lines. (Default: 0.3)

contour_color_levels: List[float], optional Contour color levels. (Default: *numpy.arange(0, contour_line_levels[-1] + 1, fineness)*)

fineness: float, optional Spacing of contour color levels. (Default: 0.1)

lines: bool, optional (Default: True)

rasterize_contours: bool, optional Rasterize the contours while keeping the lines, text etc in vector format. Useful for reducing file size bloat and making printing easier when you have dense contours. (Default: False)

Returns

cbar: color bar `matplotlib.contour.QuadContourSet` Colors to create a global colour bar

`fgivenx.plot.`**`plot_lines`**(*x*, *fsamps*, *ax=None*, *downsample=100*, ***kwargs*)

Plot function samples as a set of line plots.

Parameters

x: 1D array-like x values to plot

fsamps: 2D array-like set of functions to plot at each x. As returned by `fgivenx.compute_samples()`

ax: axes object `matplotlib.pyplot.ax` to plot on.

downsample: int, optional Reduce the number of samples to a viewable quantity. (Default 100)

any other keywords are passed to :meth:'matplotlib.pyplot.ax.plot'

9.9 fgivenx.samples module

`fgivenx.samples.`**`compute_samples`**(*f*, *x*, *samples*, ***kwargs*)

Apply f(x,theta) to x array and theta in samples.

Parameters

f: function list of functions $f(x;\theta)$ with dependent variable x, parameterised by θ.

x: 1D array-like x values to evaluate $f(x;\theta)$ at.

samples: 2D array-like list of theta samples to evaluate $f(x;\theta)$ at. *shape = (nfunc, nsamples, npars)*

parallel, tqdm_kwargs: optional see docstring for *fgivenx.parallel.parallel_apply()*

cache: str, optional File root for saving previous calculations for re-use default None

Returns

2D numpy.array: samples at each x. *shape=(len(x),len(samples),)*

`fgivenx.samples.`**`samples_from_getdist_chains`**(*params*, *file_root*, *latex=False*, ***kwargs*)

Extract samples and weights from getdist chains.

Parameters

params: list(str) Names of parameters to be supplied to second argument of f(x|theta).

file_root: str, optional Root name for getdist chains files. This variable automatically defines: - chains_file = file_root.txt - paramnames_file = file_root.paramnames but can be overidden by chains_file or paramnames_file.

latex: bool, optional Also return an array of latex strings for those paramnames.

Any additional keyword arguments are forwarded onto getdist, e.g:

samples_from_getdist_chains(params, file_root, settings={'ignore_rows':0.5})

Returns

samples: numpy.array 2D Array of samples. *shape=(len(samples), len(params))*

weights: numpy.array Array of weights. *shape = (len(params),)*

latex: list(str), optional list of latex strings for each parameter (if latex is provided as an argument)

CHAPTER 10

fgivenx: Functional Posterior Plotter

fgivenx Functional Posterior Plotter

Author Will Handley

Version 2.2.1

Homepage https://github.com/williamjameshandley/fgivenx

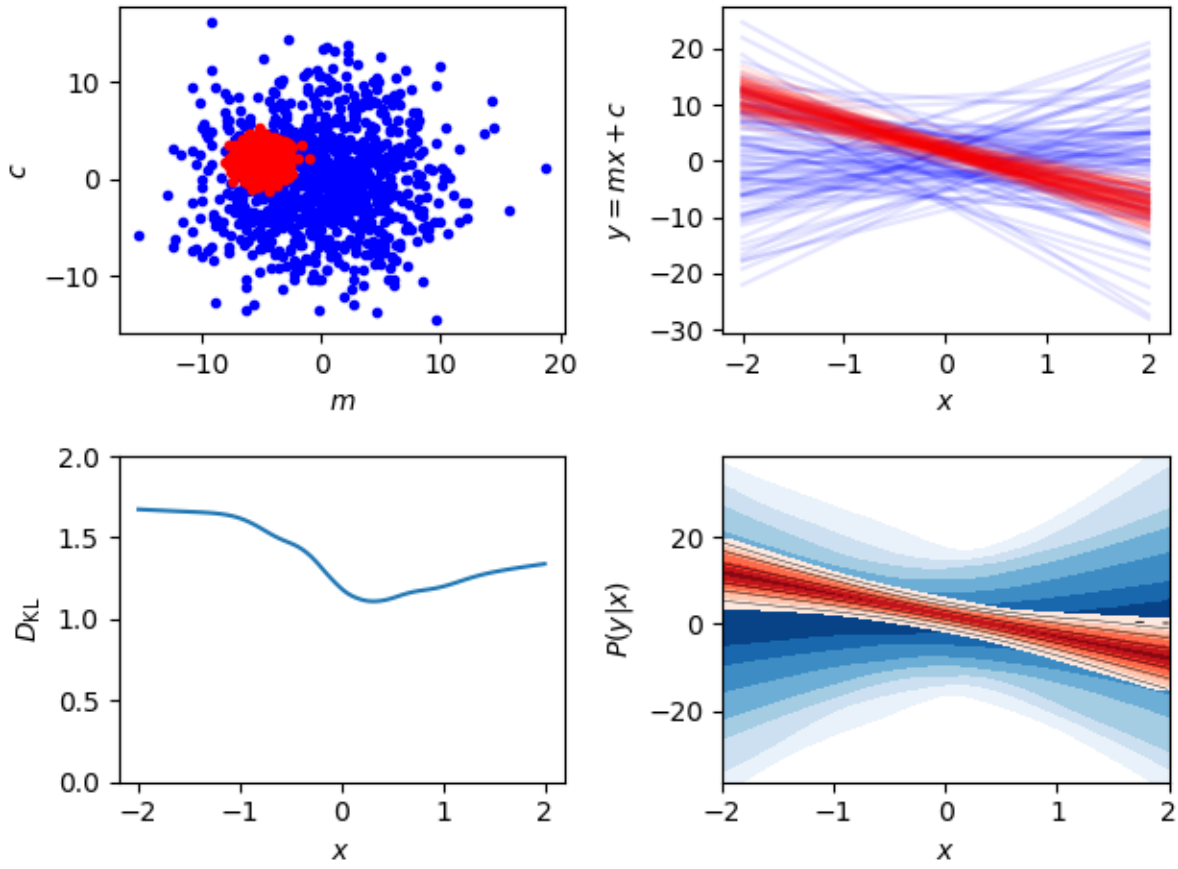
Documentation http://fgivenx.readthedocs.io/

10.1 Description

`fgivenx` is a python package for plotting posteriors of functions. It is currently used in astronomy, but will be of use to any scientists performing Bayesian analyses which have predictive posteriors that are functions.

This package allows one to plot a predictive posterior of a function, dependent on sampled parameters. We assume one has a Bayesian posterior `Post(theta|D,M)` described by a set of posterior samples `{theta_i}~Post`. If there is a function parameterised by theta `y=f(x;theta)`, then this script will produce a contour plot of the conditional posterior `P(y|x,D,M)` in the `(x,y)` plane.

The driving routines are `fgivenx.plot_contours`, `fgivenx.plot_lines` and `fgivenx.plot_dkl`. The code is compatible with getdist, and has a loading function provided by `fgivenx.samples_from_getdist_chains`.



10.2 Getting Started

Users can install using pip:

```
pip install fgivenx
```

from source:

```
git clone https://github.com/williamjameshandley/fgivenx
cd fgivenx
python setup.py install --user
```

or for those on Arch linux it is available on the AUR

You can check that things are working by running the test suite (You may encounter warnings if the optional dependency `joblib` is not installed):

```
pip install pytest pytest-runner pytest-mpl
export MPLBACKEND=Agg
pytest <fgivenx-install-location>
```

(continues on next page)

(continued from previous page)

```
# or, equivalently
git clone https://github.com/williamjameshandley/fgivenx
cd fgivenx
python setup.py test
```

Check the dependencies listed in the next section are installed. You can then use the `fgivenx` module from your scripts.

Some users of OSX or Anaconda may find `QueueManagerThread` errors if Pillow is not installed (run `pip install pillow`).

If you want to use parallelisation, have progress bars or getdist compatibility you should install the additional optional dependencies:

```
pip install joblib tqdm getdist
# or, equivalently
pip install -r  requirements.txt
```

You may encounter warnings if you don't have the optional dependency `joblib` installed.

10.3 Dependencies

Basic requirements:

- Python 2.7+ or 3.4+
- matplotlib
- numpy
- scipy

Documentation:

- sphinx
- numpydoc

Tests:

- pytest
- pytest-mpl

Optional extras:

- joblib (parallelisation) [+ pillow on some systems]
- tqdm (progress bars)
- getdist (reading of getdist compatible files)

10.4 Documentation

Full Documentation is hosted at ReadTheDocs. To build your own local copy of the documentation you'll need to install sphinx. You can then run:

```
cd docs
make html
```

10.5 Citation

If you use `fgivenx` to generate plots for a publication, please cite as:

```
Handley, (2018). fgivenx: A Python package for functional posterior
plotting . Journal of Open Source Software, 3(28), 849,
https://doi.org/10.21105/joss.00849
```

or using the BibTeX:

```
@article{fgivenx,
    doi = {10.21105/joss.00849},
    url = {http://dx.doi.org/10.21105/joss.00849},
    year  = {2018},
    month = {Aug},
    publisher = {The Open Journal},
    volume = {3},
    number = {28},
    author = {Will Handley},
    title = {fgivenx: Functional Posterior Plotter},
    journal = {The Journal of Open Source Software}
}
```

10.6 Example Usage

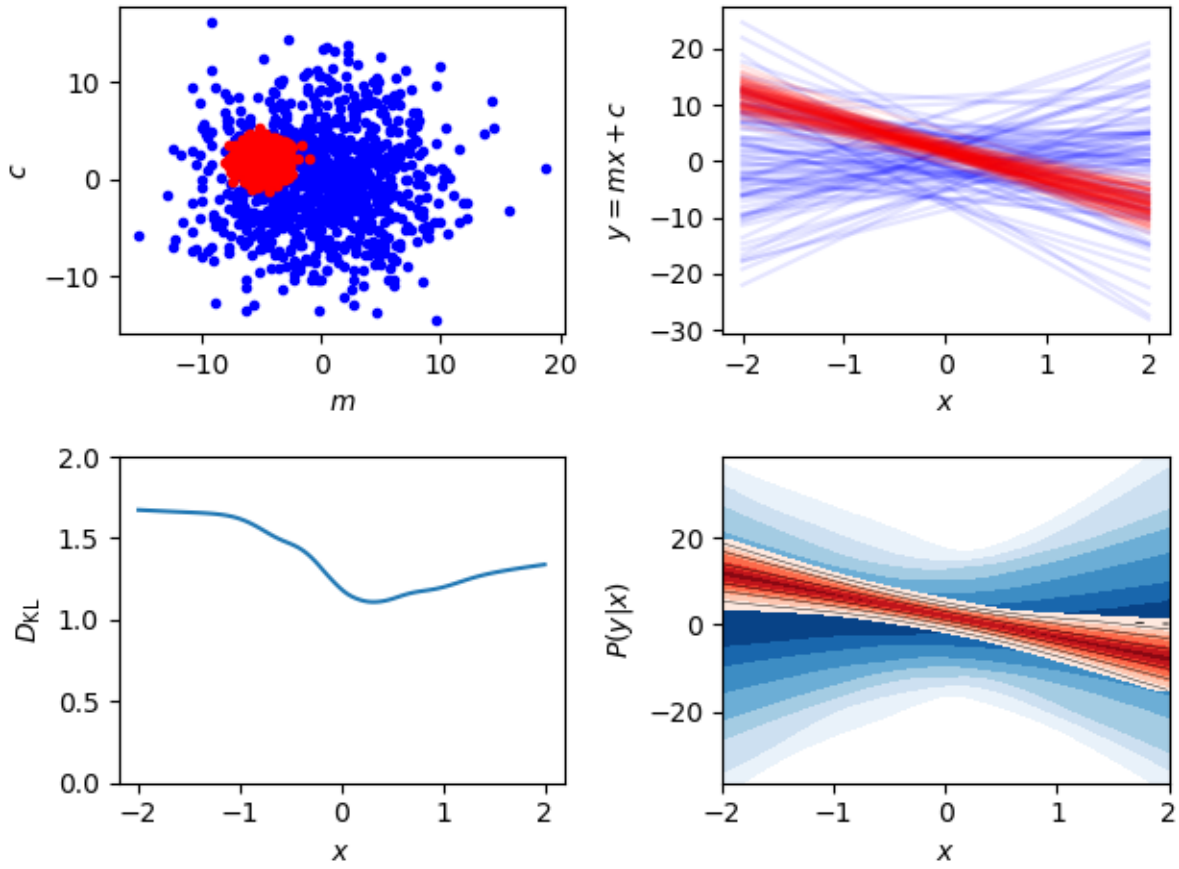
10.6.1 Plot user-generated samples

```
import numpy
import matplotlib.pyplot as plt
from fgivenx import plot_contours, plot_lines, plot_dkl


# Model definitions
# =================
# Define a simple straight line function, parameters theta=(m,c)
def f(x, theta):
    m, c = theta
    return m * x + c


numpy.random.seed(1)

# Posterior samples
nsamples = 1000
ms = numpy.random.normal(loc=-5, scale=1, size=nsamples)
cs = numpy.random.normal(loc=2, scale=1, size=nsamples)
samples = numpy.array([(m, c) for m, c in zip(ms, cs)]).copy()
```

(continues on next page)

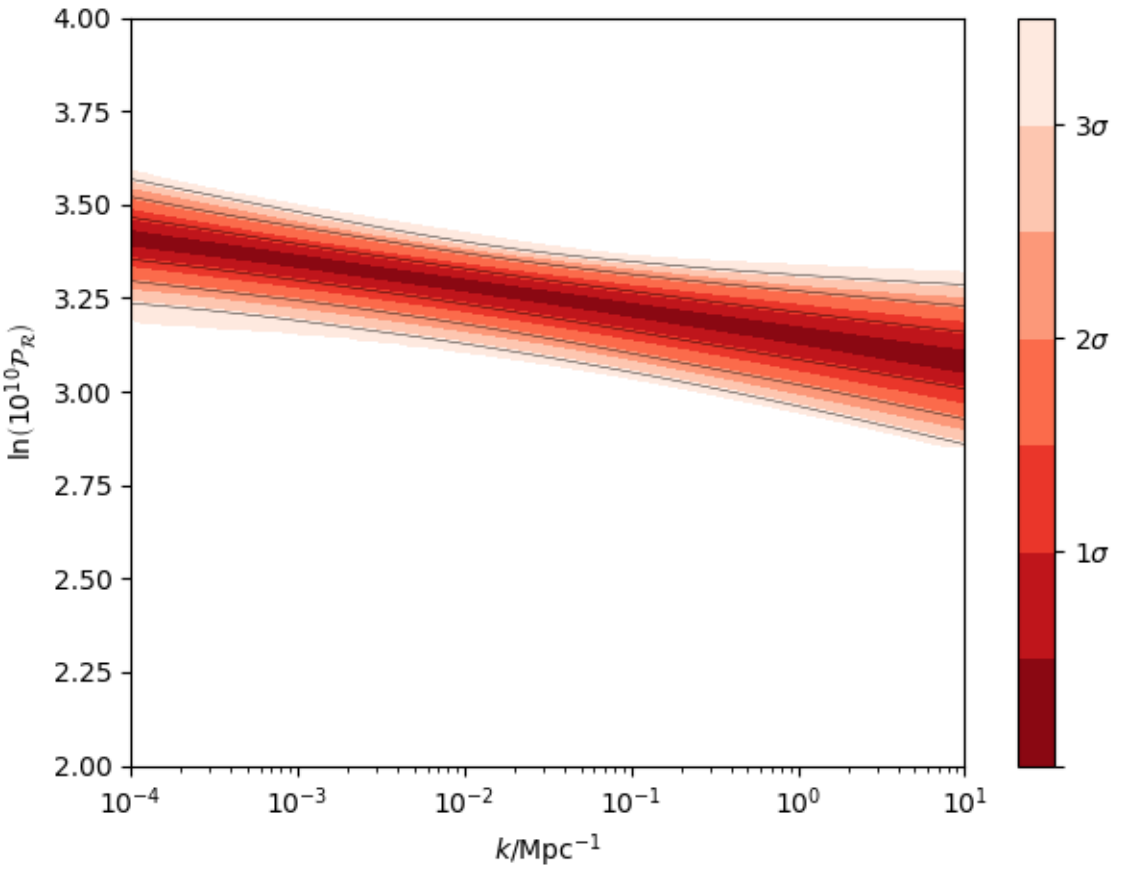
(continued from previous page)

```
# Prior samples
ms = numpy.random.normal(loc=0, scale=5, size=nsamples)
cs = numpy.random.normal(loc=0, scale=5, size=nsamples)
prior_samples = numpy.array([(m, c) for m, c in zip(ms, cs)]).copy()

# Set the x range to plot on
xmin, xmax = -2, 2
nx = 100
x = numpy.linspace(xmin, xmax, nx)

# Set the cache
cache = 'cache/test'
prior_cache = cache + '_prior'

# Plotting
# ========
fig, axes = plt.subplots(2, 2)

# Sample plot
# -----------
ax_samples = axes[0, 0]
ax_samples.set_ylabel(r'$c$')
ax_samples.set_xlabel(r'$m$')
ax_samples.plot(prior_samples.T[0], prior_samples.T[1], 'b.')
ax_samples.plot(samples.T[0], samples.T[1], 'r.')

# Line plot
# ---------
ax_lines = axes[0, 1]
ax_lines.set_ylabel(r'$y = m x + c$')
ax_lines.set_xlabel(r'$x$')
plot_lines(f, x, prior_samples, ax_lines, color='b', cache=prior_cache)
plot_lines(f, x, samples, ax_lines, color='r', cache=cache)

# Predictive posterior plot
# -------------------------
ax_fgivenx = axes[1, 1]
ax_fgivenx.set_ylabel(r'$P(y|x)$')
ax_fgivenx.set_xlabel(r'$x$')
cbar = plot_contours(f, x, prior_samples, ax_fgivenx,
                     colors=plt.cm.Blues_r, lines=False,
                     cache=prior_cache)
cbar = plot_contours(f, x, samples, ax_fgivenx, cache=cache)

# DKL plot
# --------
ax_dkl = axes[1, 0]
ax_dkl.set_ylabel(r'$D_\mathrm{KL}$')
ax_dkl.set_xlabel(r'$x$')
ax_dkl.set_ylim(bottom=0, top=2.0)
plot_dkl(f, x, samples, prior_samples, ax_dkl,
         cache=cache, prior_cache=prior_cache)

ax_lines.get_shared_x_axes().join(ax_lines, ax_fgivenx, ax_samples)

fig.tight_layout()
fig.savefig('plot.png')
```



10.6.2 Plot GetDist chains

```
import numpy
import matplotlib.pyplot as plt
from fgivenx import plot_contours, samples_from_getdist_chains

file_root = './plik_HM_TT_lowl/base_plikHM_TT_lowl'
samples, weights = samples_from_getdist_chains(['logA', 'ns'], file_root)

def PPS(k, theta):
    logA, ns = theta
    return logA + (ns - 1) * numpy.log(k)

k = numpy.logspace(-4,1,100)
cbar = plot_contours(PPS, k, samples, weights=weights)
cbar = plt.colorbar(cbar,ticks=[0,1,2,3])
cbar.set_ticklabels(['',r'$1\sigma$',r'$2\sigma$',r'$3\sigma$'])

plt.xscale('log')
plt.ylim(2,4)
plt.ylabel(r'$\ln\left(10^{10}\mathcal{P}_\mathcal{R}\right)$')
plt.xlabel(r'$k / {\rm Mpc}^{-1}$')
plt.tight_layout()
plt.savefig('planck.png')
```



10.7 Contributing

Want to contribute to `fgivenx`? Awesome! There are many ways you can contribute via the [GitHub repository](https://github.com/williamjameshandley/fgivenx), see below.

10.7.1 Opening issues

Open an issue to report bugs or to propose new features.

10.7.2 Proposing pull requests

Pull requests are very welcome. Note that if you are going to propose drastic changes, be sure to open an issue for discussion first, to make sure that your PR will be accepted before you spend effort coding it.

10.8 Changelog

v2.2.0 Paper accepted

v2.1.17 100% coverage

v2.1.16 Tests fixes

v2.1.15 Additional plot tests

v2.1.13 Further bug fix in test suite for image comparison

v2.1.12 Bug fix in test suite for image comparison

v2.1.11 Documentation upgrades

v2.1.10 Added changelog

Python Module Index

f

Index