Fork me on GitHub

Example: Fitting a Model to Data

If you're reading this right now then you're probably interested in using emcee to fit a model to some noisy data. On this page, I'll demonstrate how you might do this in the simplest non-trivial model that I could think of: fitting a line to data when you don't believe the error bars on your data. The interested reader should check out Hogg, Bovy & Lang (2010) for a much more complete discussion of how to fit a line to data in The Real World™ and why MCMC might come in handy.

The full source code for this example is available in the GitHub repository.

The generative probabilistic model

When you approach a new problem, the first step is generally to write down the *likelihood function* (the probability of a dataset given the model parameters). This is equivalent to describing the generative procedure for the data. In this case, we're going to consider a linear model where the quoted uncertainties are underestimated by a constant fractional amount. You can generate a synthetic dataset from this model:

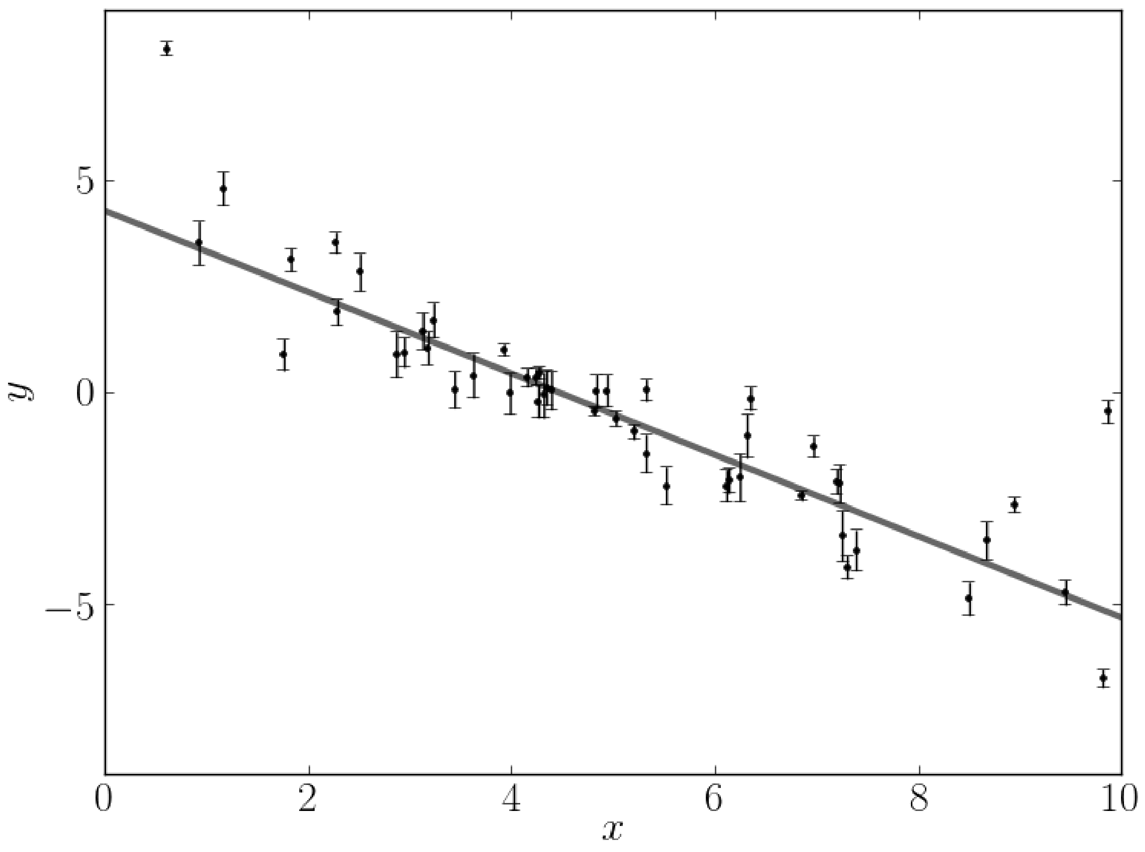
```
import numpy as np

# Choose the "true" parameters.
m_true = -0.9594
b_true = 4.294
f_true = 0.534

# Generate some synthetic data from the model.
N = 50
x = np.sort(10*np.random.rand(N))
yerr = 0.1+0.5*np.random.rand(N)
y = m_true*x+b_true
y += np.abs(f_true*y) * np.random.randn(N)
y += yerr * np.random.randn(N)
```

This synthetic dataset (with the underestimated error bars) will look something like:

v: stable



The true model is shown as the thick grey line and the effect of the underestimated uncertainties is obvious when you look at this figure. The standard way to fit a line to these data (assuming independent Gaussian error bars) is linear least squares. Linear least squares is appealing because solving for the parameters—and their associated uncertainties—is simply a linear algebraic operation. Following the notation in Hogg, Bovy & Lang (2010), the linear least squares solution to these data is

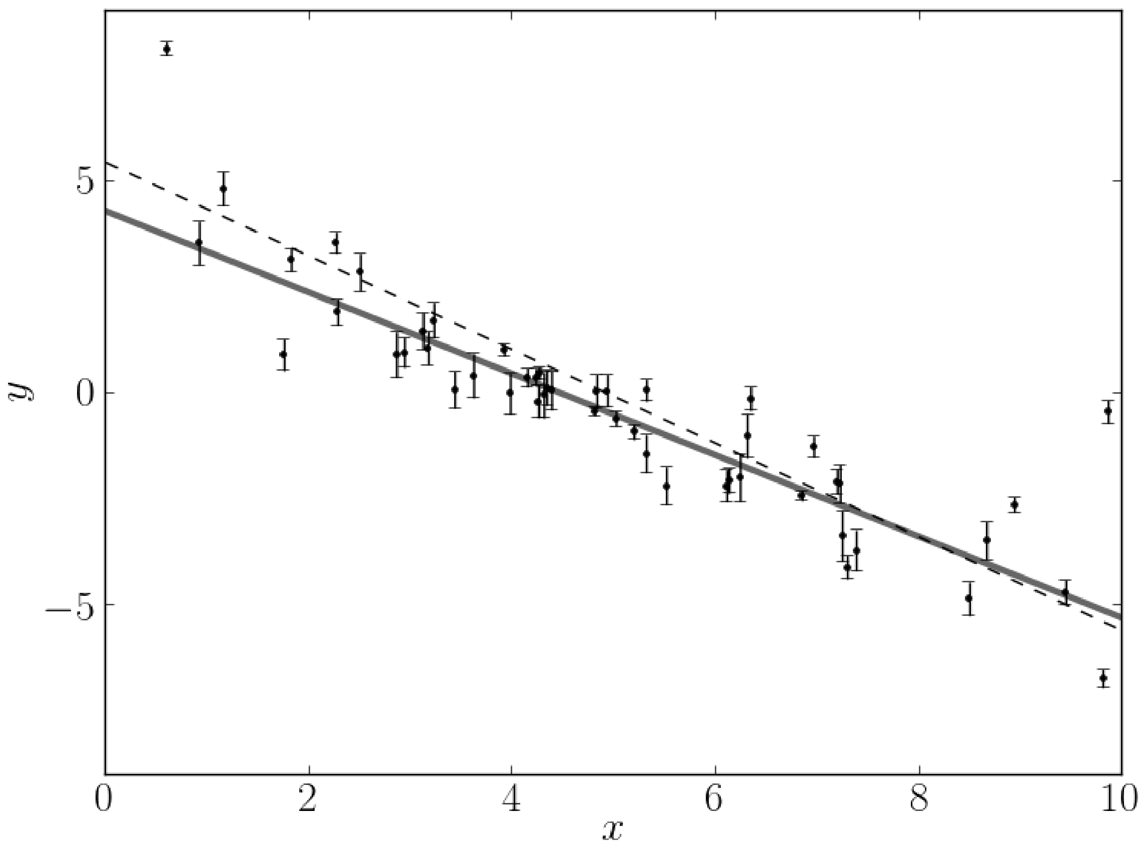
```
A = np.vstack((np.ones_like(x), x)).T
C = np.diag(yerr * yerr)
cov = np.linalg.inv(np.dot(A.T, np.linalg.solve(C, A)))
b_ls, m_ls = np.dot(cov, np.dot(A.T, np.linalg.solve(C, y)))
```

For the dataset generated above, the result is

$$m = -1.104 \pm 0.016 \quad \text{and} \quad b = 5.441 \pm 0.091$$

plotted below as a dashed line:

v: stable



This isn't an unreasonable result but the uncertainties on the slope and intercept seem a little small (because of the small error bars on most of the data points).

Maximum likelihood estimation

The least squares solution found in the previous section is the maximum likelihood result for a model where the error bars are assumed correct, Gaussian and independent. We know, of course, that this isn't the right model. Unfortunately, there isn't a generalization of least squares that supports a model like the one that we know to be true. Instead, we need to write down the likelihood function and numerically optimize it. In mathematical notation, the correct likelihood function is:

$$\ln\, p(y\,|\,x,\sigma,m,b,f) = -\frac{1}{2}\sum_n \left[\frac{(y_n - m\,x_n - b)^2}{s_n^2} + \ln\left(2\pi\,s_n^2\right)\right]$$

where

$$s_n^2 = \sigma_n^2 + f^{\,2}\,(m\,x_n + b)^2 \quad .$$

This likelihood function is simply a Gaussian where the variance is underestimated by some fractional amount: *f*. In Python, you would code this up as:

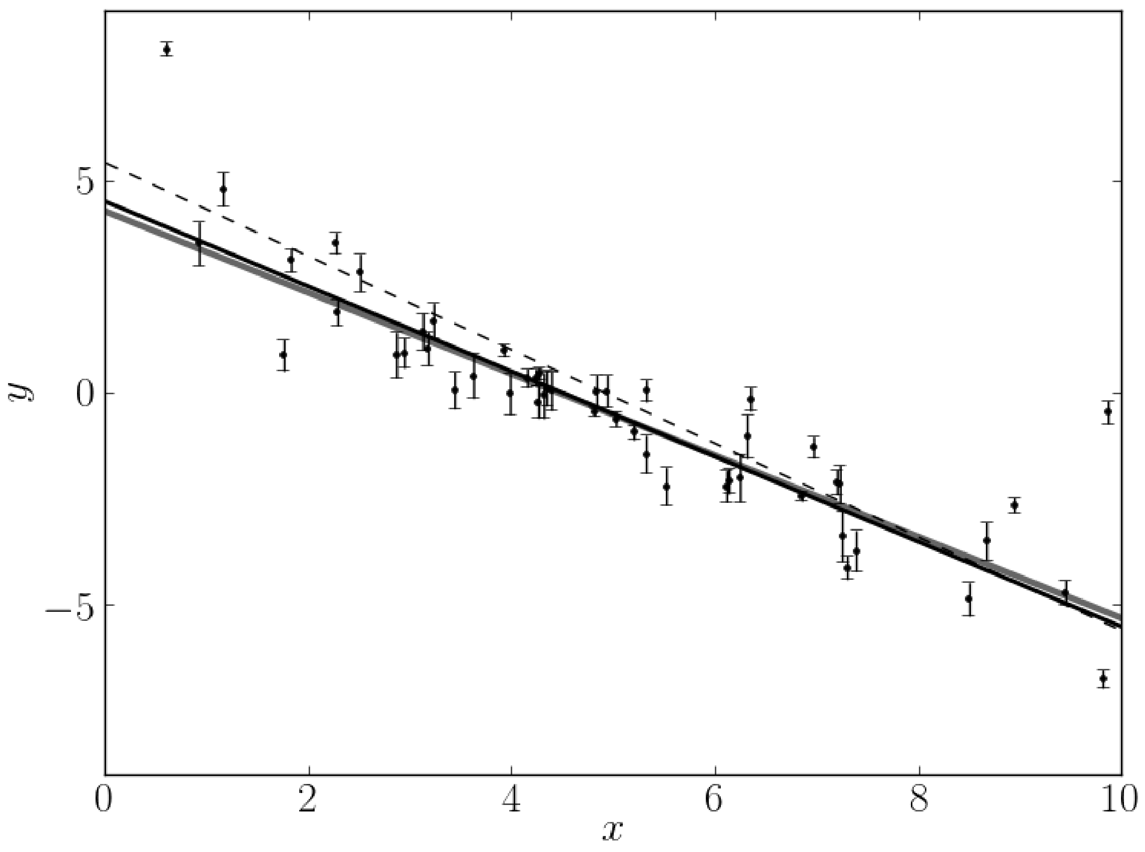
```python
def lnlike(theta, x, y, yerr):
    m, b, lnf = theta
    model = m * x + b
    inv_sigma2 = 1.0/(yerr**2 + model**2*np.exp(2*lnf))
    return -0.5*(np.sum((y-model)**2*inv_sigma2 - np.log(inv_sigma2)))
```

In this code snippet, you'll notice that I'm using the logarithm of *f* instead of *f* itself for reasons that will become clear in the next section. For now, it should at least be clear that this isn't a bad idea because it will force *f* to be always positive. A good way of finding this numerical optimum of this likelihood function is to use the scipy.optimize module:

v: stable

```
import scipy.optimize as op
nll = lambda *args: -lnlike(*args)
result = op.minimize(nll, [m_true, b_true, np.log(f_true)], args=(x, y, yerr))
m_ml, b_ml, lnf_ml = result["x"]
```

It's worth noting that the optimize module *minimizes* functions whereas we would like to maximize the likelihood. This goal is equivalent to minimizing the *negative* likelihood (or in this case, the negative *log* likelihood). The maximum likelihood result is plotted as a solid black line—compared to the true model (grey line) and linear least squares (dashed line)—in the following figure:



That looks better! The values found by this optimization are:

$$m = -1.003\,, \quad b = 4.528 \quad \text{and} \quad f = 0.454 \quad .$$

The problem now: how do we estimate the uncertainties on *m* and *b*? What's more, we probably don't really care too much about the value of *f* but it seems worthwhile to propagate any uncertainties about its value to our final estimates of *m* and *b*. This is where MCMC comes in.

Marginalization & uncertainty estimation

This isn't the place to get into the details of why you might want to use MCMC in your research but it is worth commenting that a common reason is that you would like to marginalize over some "nuisance parameters" and find an estimate of the posterior probability function (the distribution of parameters that is consistent with your dataset) for others. MCMC lets you do both of these things in one fell swoop! You need to start by writing down the posterior probability function (up to a constant):

$$p(m, b, f \mid x, y, \sigma) \propto p(m, b, f)\, p(y \mid x, \sigma, m, b, f) \quad .$$

We have already, in the previous section, written down the likelihood function

$$p(y \mid x, \sigma, m, b, f)$$

v: stable

so the missing component is the “prior” function

$$p(m, b, f) \quad .$$

This function encodes any previous knowledge that we have about the parameters: results from other experiments, physically acceptable ranges, etc. It is necessary that you write down priors if you’re going to use MCMC because all that MCMC does is draw samples from a probability distribution and you want that to be a probability distribution for your parameters. This is important: **you cannot draw parameter samples from your likelihood function**. This is because a likelihood function is a probability distribution **over datasets** so, conditioned on model parameters, you can draw representative datasets (as demonstrated at the beginning of this exercise) but you cannot draw parameter samples.

In this example, we’ll use uniform (so-called “uninformative”) priors on m, b and the logarithm of f. For example, we’ll use the following conservative prior on m:

$$p(m) = \begin{cases} 1/5.5\,, & \text{if } -5 < m < 1/2 \\ 0\,, & \text{otherwise} \end{cases}$$

In code, the log-prior is (up to a constant):

```
def lnprior(theta):
    m, b, lnf = theta
    if -5.0 < m < 0.5 and 0.0 < b < 10.0 and -10.0 < lnf < 1.0:
        return 0.0
    return -np.inf
```

Then, combining this with the definition of `lnlike` from above, the full log-probability function is:

```
def lnprob(theta, x, y, yerr):
    lp = lnprior(theta)
    if not np.isfinite(lp):
        return -np.inf
    return lp + lnlike(theta, x, y, yerr)
```

After all this setup, it’s easy to sample this distribution using `emcee`. We’ll start by initializing the walkers in a tiny Gaussian ball around the maximum likelihood result (I’ve found that this tends to be a pretty good initialization in most cases):

```
ndim, nwalkers = 3, 100
pos = [result["x"] + 1e-4*np.random.randn(ndim) for i in range(nwalkers)]
```

Then, we can set up the sampler:

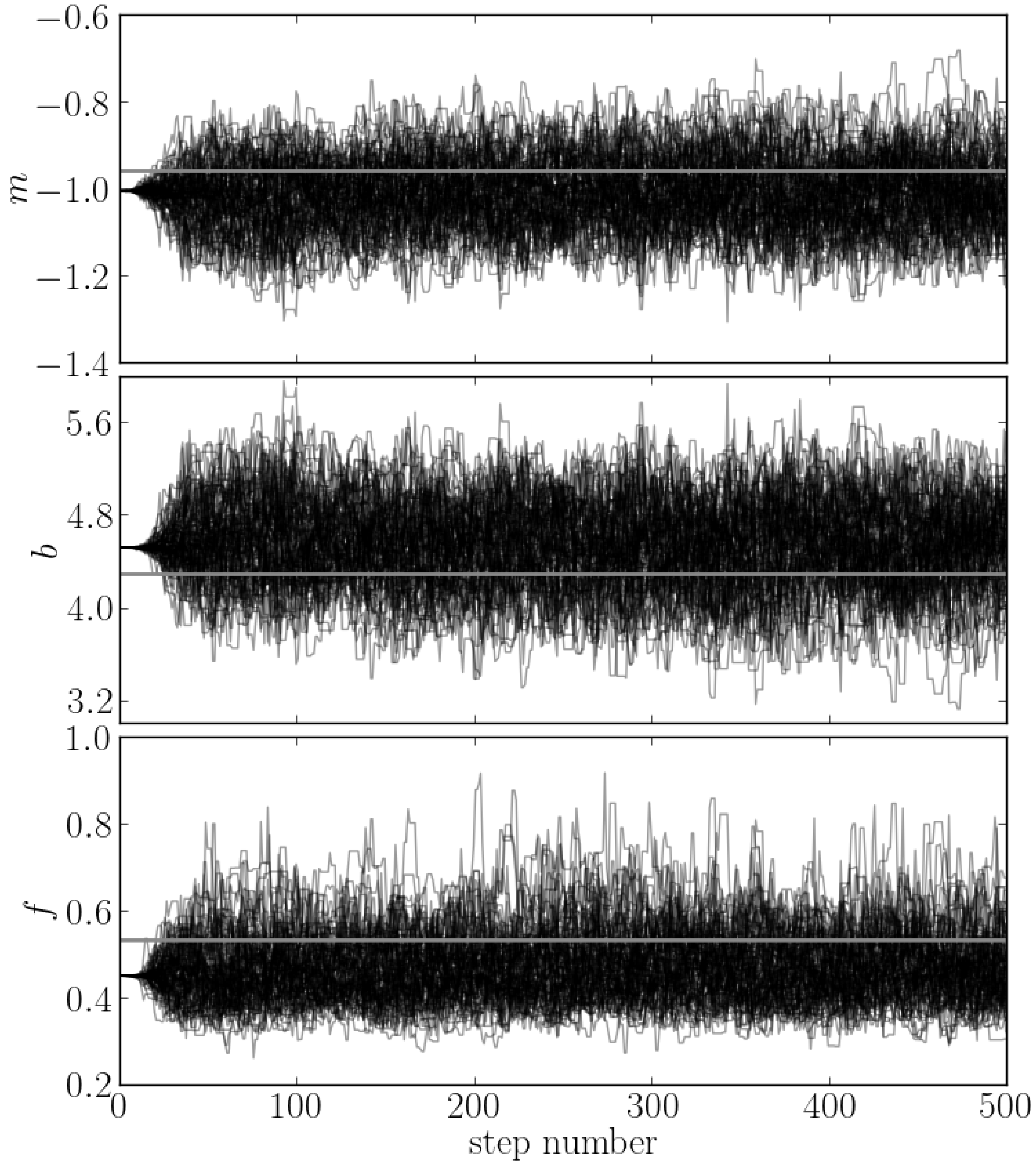
```
import emcee
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob, args=(x, y, yerr))
```

and run the MCMC for 500 steps starting from the tiny ball defined above:

```
sampler.run_mcmc(pos, 500)
```

Let’s take a look at what the sampler has done. The best way to see this is to look at the time series of the parameters in the chain. The `sampler` object now has an attribute called `chain` that is an array with the shape `(100, 500, 3)` giving the parameter values for each walker at each step in the chain. The figure below shows the positions of each walker as a function of the number of steps in the chain:

v: stable



The true values of the parameters are indicated as grey lines on top of the samples. As mentioned above, the walkers start in small distributions around the maximum likelihood values and then they quickly wander and start exploring the full posterior distribution. In fact, after fewer than 50 steps, the samples seem pretty well "burnt-in". That is a hard statement to make quantitatively but for now, we'll just accept it and discard the initial 50 steps and flatten the chain so that we have a flat list of samples:

```
samples = sampler.chain[:, 50:, :].reshape((-1, ndim))
```

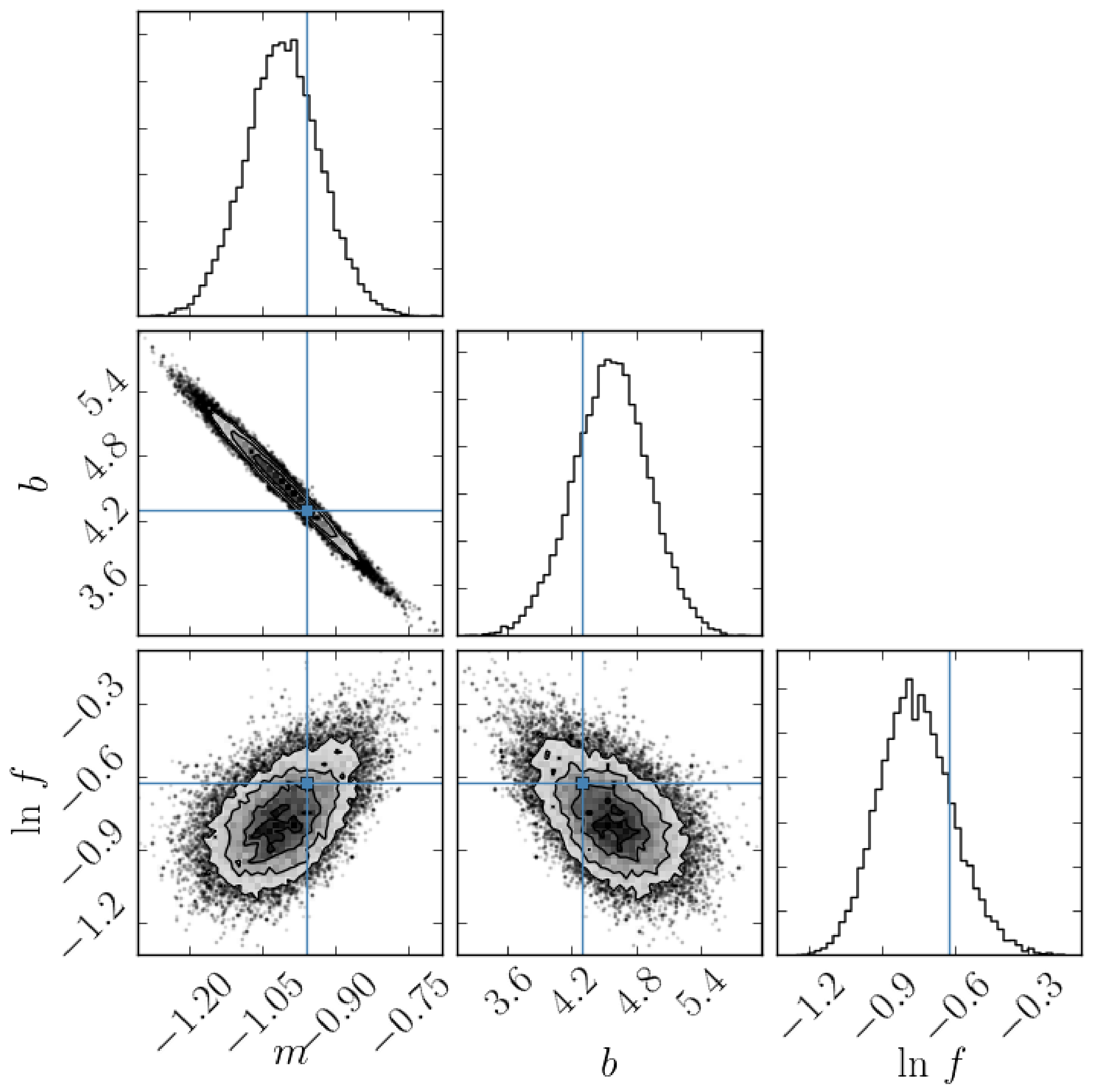
Results

Now that we have this list of samples, let's make one of the most useful plots you can make with your MCMC results: *a corner plot*. You'll need the corner.py module but once you have it, generating a corner plot is as simple as:

```
import corner
fig = corner.corner(samples, labels=["$m$", "$b$", "$\ln\,f$"],
                      truths=[m_true, b_true, np.log(f_true)])
fig.savefig("triangle.png")
```

v: stable

and you should get something like the following:



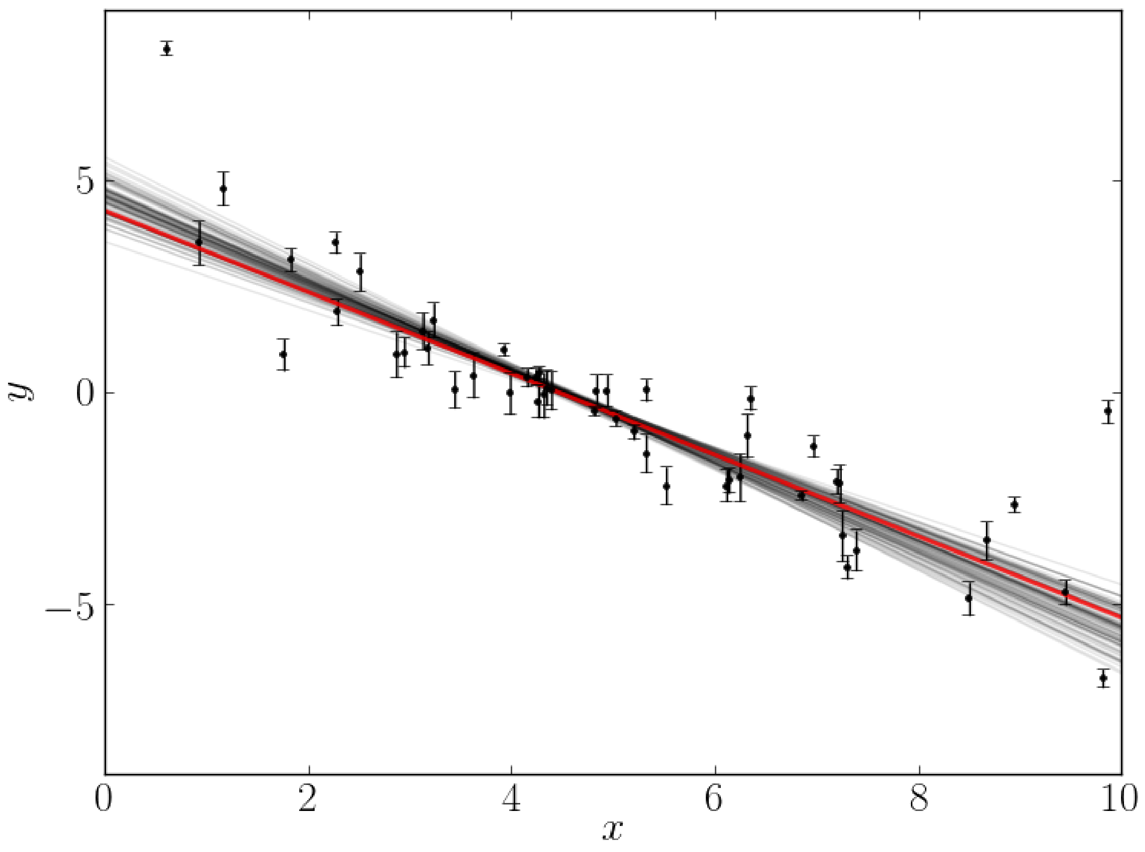
The corner plot shows all the one and two dimensional projections of the posterior probability distributions of your parameters. This is useful because it quickly demonstrates all of the covariances between parameters. Also, the way that you find the marginalized distribution for a parameter or set of parameters using the results of the MCMC chain is to project the samples into that plane and then make an N-dimensional histogram. That means that the corner plot shows the marginalized distribution for each parameter independently in the histograms along the diagonal and then the marginalized two dimensional distributions in the other panels.

Another diagnostic plot is the projection of your results into the space of the observed data. To do this, you can choose a few (say 100 in this case) samples from the chain and plot them on top of the data points:

```
import matplotlib.pyplot as pl
xl = np.array([0, 10])
for m, b, lnf in samples[np.random.randint(len(samples), size=100)]:
    pl.plot(xl, m*xl+b, color="k", alpha=0.1)
pl.plot(xl, m_true*xl+b_true, color="r", lw=2, alpha=0.8)
pl.errorbar(x, y, yerr=yerr, fmt=".k")
```

which should give you something like:

v: stable



This leaves us with one question: which numbers should go in the abstract? There are a few different options for this but my favorite is to quote the uncertainties based on the 16th, 50th, and 84th percentiles of the samples in the marginalized distributions. To compute these numbers for this example, you would run:

```
samples[:, 2] = np.exp(samples[:, 2])
m_mcmc, b_mcmc, f_mcmc = map(lambda v: (v[1], v[2]-v[1], v[1]-v[0]),
                             zip(*np.percentile(samples, [16, 50, 84],
                                                axis=0)))
```

giving you the results:

$$m = -1.009^{+0.077}_{-0.075}\,, \quad b = 4.556^{+0.346}_{-0.353} \quad \text{and} \quad f = 0.463^{+0.079}_{-0.063}$$

which isn't half bad given the true values:

$$m_{\text{true}} = -0.9594\,, \quad b_{\text{true}} = 4.294 \quad \text{and} \quad f_{\text{true}} = 0.534 \quad .$$

v: stable