emcee Documentation

Release 2.2.1

Dan Foreman-Mackey

Sep 22, 2017

Contents

CHAPTER 1

Seriously Kick-Ass MCMC

emcee is an MIT licensed pure-Python implementation of Goodman & Weare's Affine Invariant Markov chain Monte Carlo (MCMC) Ensemble sampler and these pages will show you how to use it.

This documentation won't teach you too much about MCMC but there are a lot of resources available for that (try this one). We also published a paper explaining the emcee algorithm and implementation in detail.

emcee has been used in quite a few projects in the astrophysical literature and it is being actively developed on GitHub.

CHAPTER 2

Basic Usage

If you wanted to draw samples from a 10 dimensional Gaussian, you would do something like:

```
import numpy as np
import emcee

def lnprob(x, ivar):
    return -0.5 * np.sum(ivar * x ** 2)

ndim, nwalkers = 10, 100
ivar = 1. / np.random.rand(ndim)
p0 = [np.random.rand(ndim) for i in range(nwalkers)]

sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob, args=[ivar])
sampler.run_mcmc(p0, 1000)
```

A more complete example is available in the quickstart documentation.

CHAPTER 3

User Guide

Installation

Since emcee is a pure Python module, it should be pretty easy to install. All you'll need numpy. There are a bunch of different ways to install and I'll mention a few below but by far the best is to install into a virtual environment using pip.

Using pip

The easiest way to install the most recent stable version of emcee is with pip:

```
$ pip install emcee
```

You might need to run this using sudo depending on your Python installation. You can also use easy_install:

```
$ easy_install emcee
```

but pip is probably better.

From source

Alternatively, you can get the source by downloading a tarball:

```
$ curl -OL https://github.com/dfm/emcee/tarball/master
```

or zip archive:

```
$ curl -OL https://github.com/dfm/emcee/zipball/master
```

Once you've downloaded and unpacked the source, you can navigate into the root source directory and run:

```
$ python setup.py install
```

Bleeding edge development version

`emcee` is being developed actively on GitHub so if you feel like hacking, you can clone the source repository

```
git clone https://github.com/dfm/emcee.git
```

or fork the repository.

Test the installation

To make sure that the installation went alright, you can run some unit tests by running:

```
python -c 'import emcee; emcee.test()'
```

or, if you have nose:

```
nosetests
```

This might take a few minutes but you shouldn't get any errors if all went as planned.

Quickstart

The easiest way to get started with using `emcee` is to use it for a project. To get you started, here's an annotated, fully-functional example that demonstrates a standard usage pattern.

How to sample a multi-dimensional Gaussian

We're going to demonstrate how you might draw samples from the multivariate Gaussian density given by:

$$p(\vec{x}) \propto \exp\left[-\frac{1}{2}(\vec{x}-\vec{\mu})^{\mathrm{T}}\,\Sigma^{-1}\,(\vec{x}-\vec{\mu})\right]$$

where $\vec{\mu}$ is an N-dimensional vector position of the mean of the density and Σ is the square N-by-N covariance matrix.

The first thing that we need to do is import the necessary modules:

```
import numpy as np
import emcee
```

Then, we'll code up a Python function that returns the density $p(\vec{x})$ for specific values of $\vec{x}$, $\vec{\mu}$ and Σ^{-1}. In fact, `emcee` actually requires the logarithm of p. We'll call it `lnprob`:

```
def lnprob(x, mu, icov):
    diff = x-mu
    return -np.dot(diff,np.dot(icov,diff))/2.0
```

It is important that the first argument of the probability function is the position of a single *walker* (a N dimensional `numpy` array). The following arguments are going to be constant every time the function is called and the values come from the `args` parameter of our `EnsembleSampler` that we'll see soon.

Now, we'll set up the specific values of those "hyperparameters" in 50 dimensions:

```
ndim = 50

means = np.random.rand(ndim)

cov = 0.5 - np.random.rand(ndim ** 2).reshape((ndim, ndim))
cov = np.triu(cov)
cov += cov.T - np.diag(cov.diagonal())
cov = np.dot(cov,cov)
```

and where `cov` is Σ. Before going on, let's compute the inverse of `cov` because that's what we need in our probability function:

```
icov = np.linalg.inv(cov)
```

It's probably overkill this time but how about we use 250 *walkers*? Before we go on, we need to guess a starting point for each of the 250 walkers. This position will be a 50-dimensional vector so the initial guess should be a 250-by-50 array—or a list of 250 arrays that each have 50 elements. It's not a very good guess but we'll just guess a random number between 0 and 1 for each component:

```
nwalkers = 250
p0 = np.random.rand(ndim * nwalkers).reshape((nwalkers, ndim))
```

Now that we've gotten past all the bookkeeping stuff, we can move on to the fun stuff. The main interface provided by `emcee` is the *`EnsembleSampler`* object so let's get ourselves one of those:

```
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob, args=[means, icov])
```

Remember how our function `lnprob` required two extra arguments when it was called? By setting up our sampler with the `args` argument, we're saying that the probability function should be called as:

```
lnprob(p, means, icov)
```

where `p` is the position of a single walker. If we didn't provide any `args` parameter, the calling sequence would be `lnprob(p)` instead.

It's generally a good idea to run a few "burn-in" steps in your MCMC chain to let the walkers explore the parameter space a bit and get settled into the maximum of the density. We'll run a burn-in of 100 steps (yep, I just made that number up... it's hard to really know how many steps of burn-in you'll need before you start) starting from our initial guess `p0`:

```
pos, prob, state = sampler.run_mcmc(p0, 100)
sampler.reset()
```

You'll notice that I saved the final position of the walkers (after the 100 steps) to a variable called `pos`. You can check out what will be contained in the other output variables by looking at the documentation for the *`EnsembleSampler.run_mcmc()`* function. The call to the *`EnsembleSampler.reset()`* method clears all of the important book-keeping parameters in the sampler so that we get a fresh start. It also clears the current positions of the walkers so it's a good thing that we saved them first.

Now, we can do our production run of 1000 steps (again, this is probably overkill... it's generally very silly to take way more samples than you need to but never mind that for now):

```
sampler.run_mcmc(pos, 1000)
```

The sampler now has a property *`EnsembleSampler.chain`* that is a `numpy` array with the shape `(250, 1000, 50)`. Take note of that shape and make sure that you know where each of those numbers come from. A much more useful object is the `EnsembleSampler.flatchain` which has the shape `(250000, 50)` and contains all the samples reshaped into a flat list. You can see now that we now have 250 000 unbiased samples of the density $p(\vec{x})$. You can make histograms of these samples to get an estimate of the density that you were sampling:

```
import matplotlib.pyplot as pl

for i in range(ndim):
    pl.figure()
    pl.hist(sampler.flatchain[:,i], 100, color="k", histtype="step")
    pl.title("Dimension {0:d}".format(i))

pl.show()
```

Another good test of whether or not the sampling went well is to check the mean acceptance fraction of the ensemble using the `EnsembleSampler.acceptance_fraction()` property:

```
print("Mean acceptance fraction: {0:.3f}"
                .format(np.mean(sampler.acceptance_fraction)))
```

This number should be between approximately 0.25 and 0.5 if everything went as planned.

Well, that's it for this example. You'll find the full, unadulterated sample code for this demo here.

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

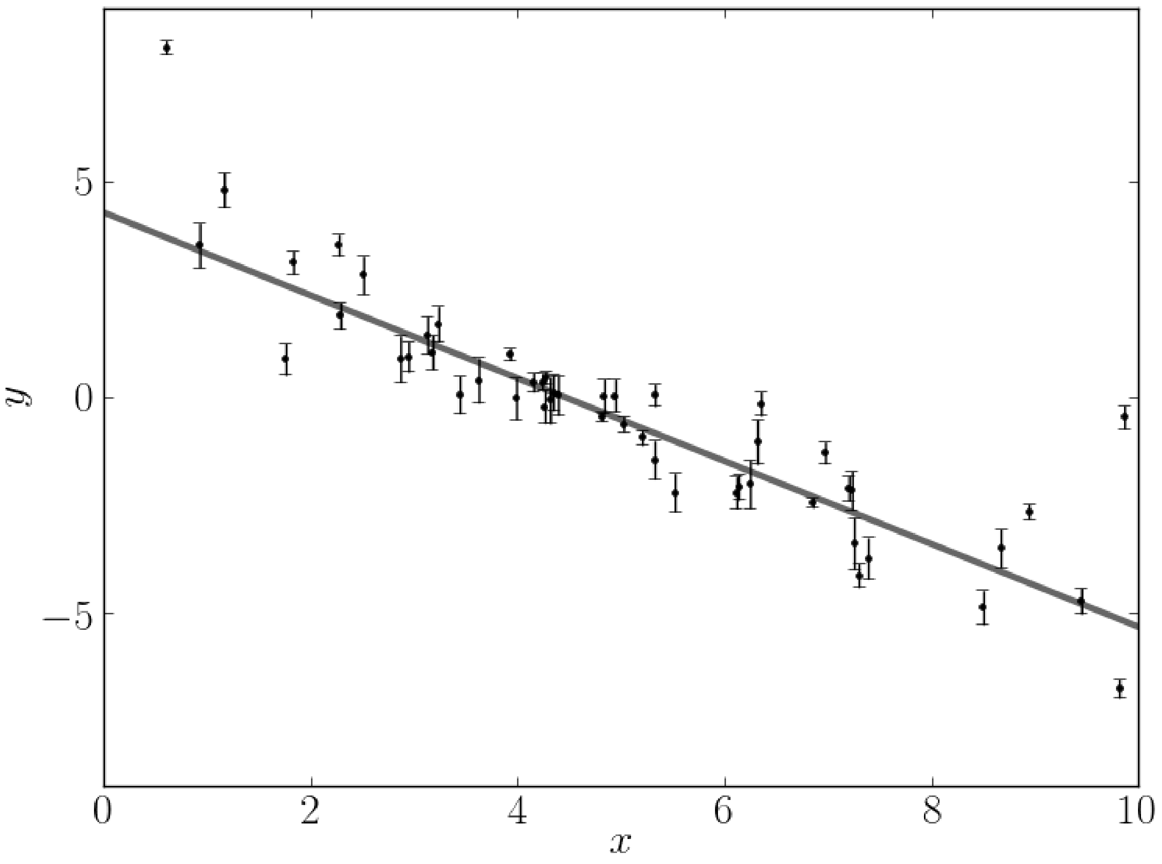
When you approach a new problem, the first step is generally to write down the *likelihood function* (the probability of a dataset given the model parameters). This is equivalent to describing the generative procedure for the data. In this case, we're going to consider a linear model where the quoted 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:



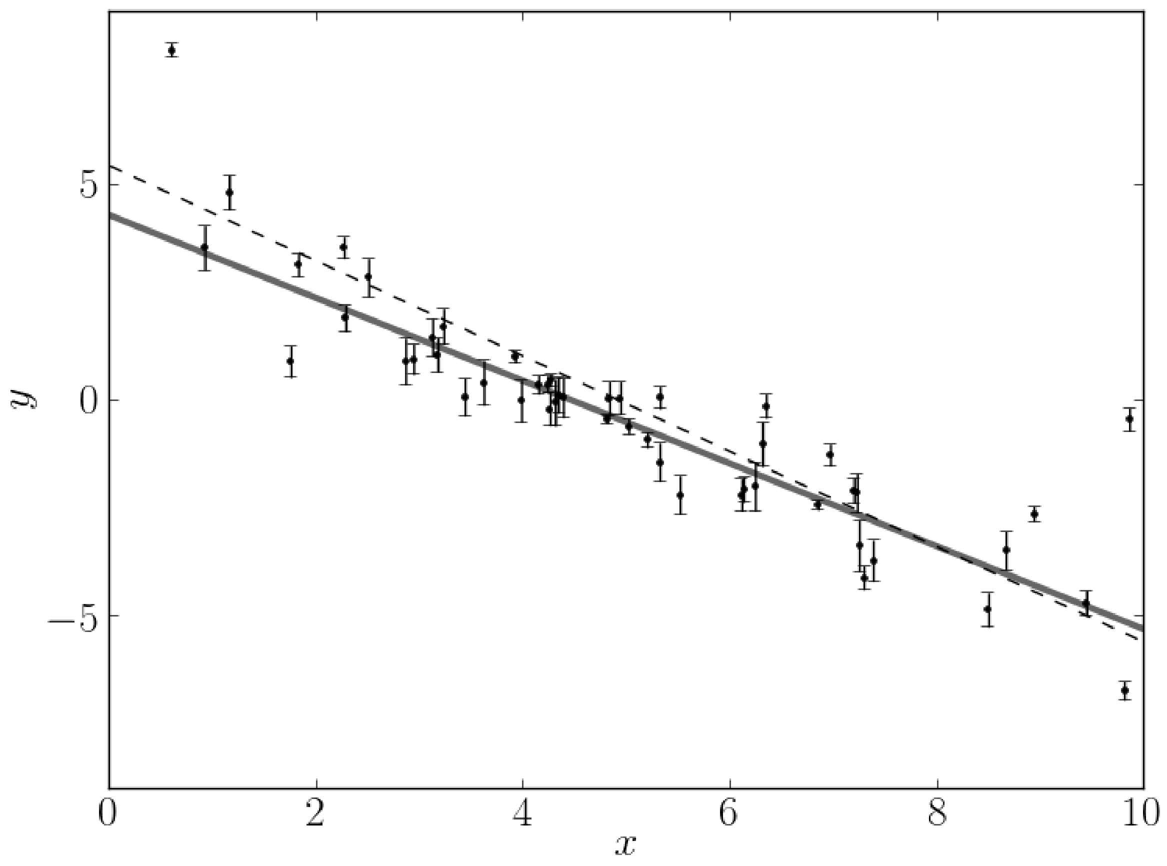
The true model is shown as the thick grey line and the effect of the underestimated uncertainties is obvious when you look at this figure. The standard way to fit a line to these data (assuming independent Gaussian error bars) is linear least squares. Linear least squares is appealing because solving 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

$$m = -1.104 \pm 0.016 \quad \text{and} \quad b = 5.4410.091$$

plotted below as a dashed line:



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

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

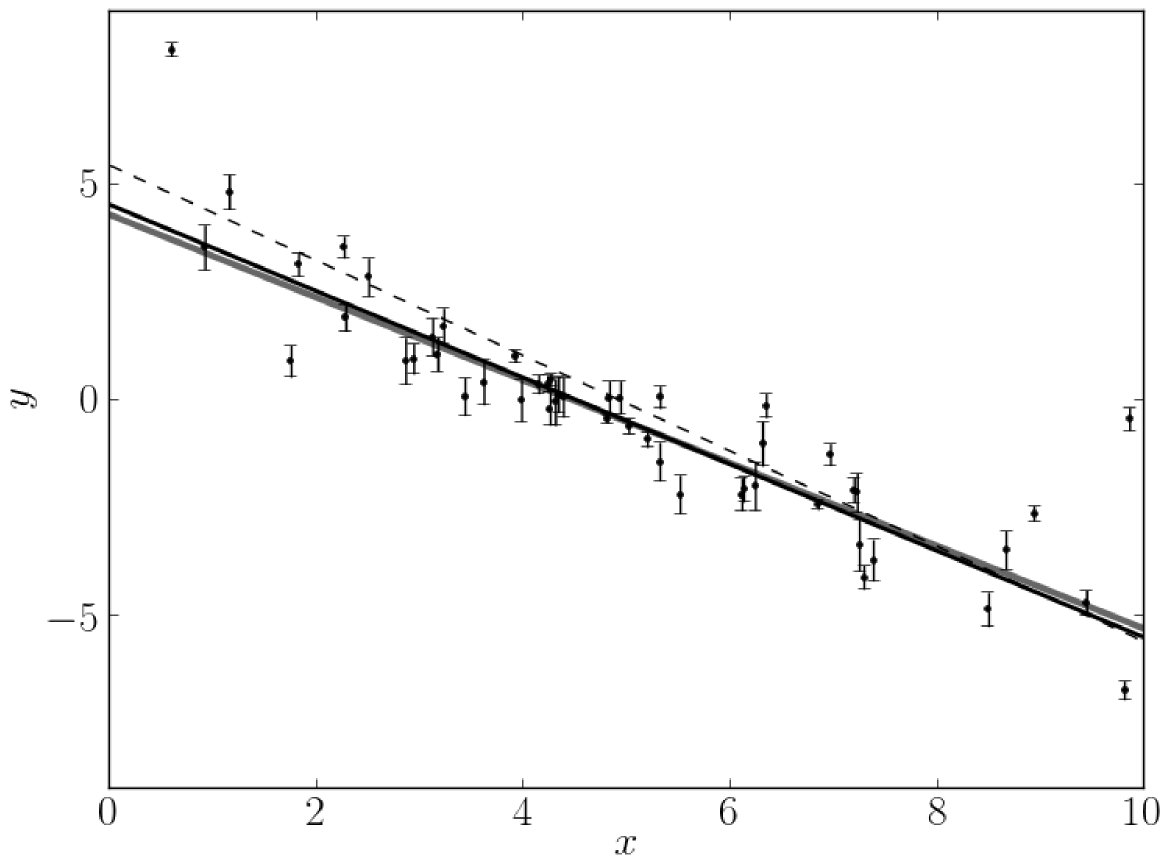
This likelihood function is simply a Gaussian where the variance is underestimated by some fractional amount: *f*. In Python, you would code this up as:

```
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:

```
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):

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

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

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

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):

```python
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:

```python
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):

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

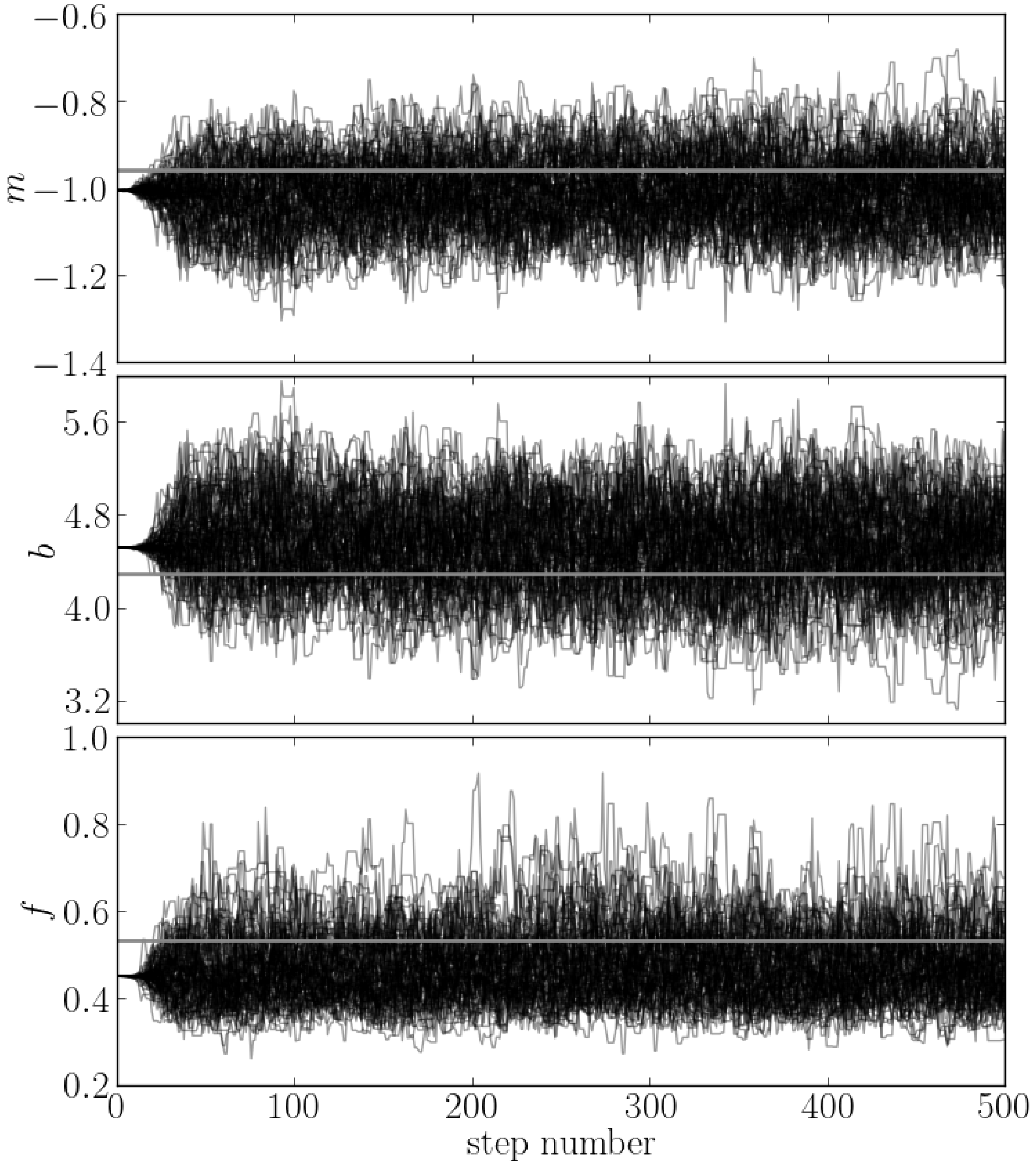
Then, we can set up the sampler:

```
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:



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:

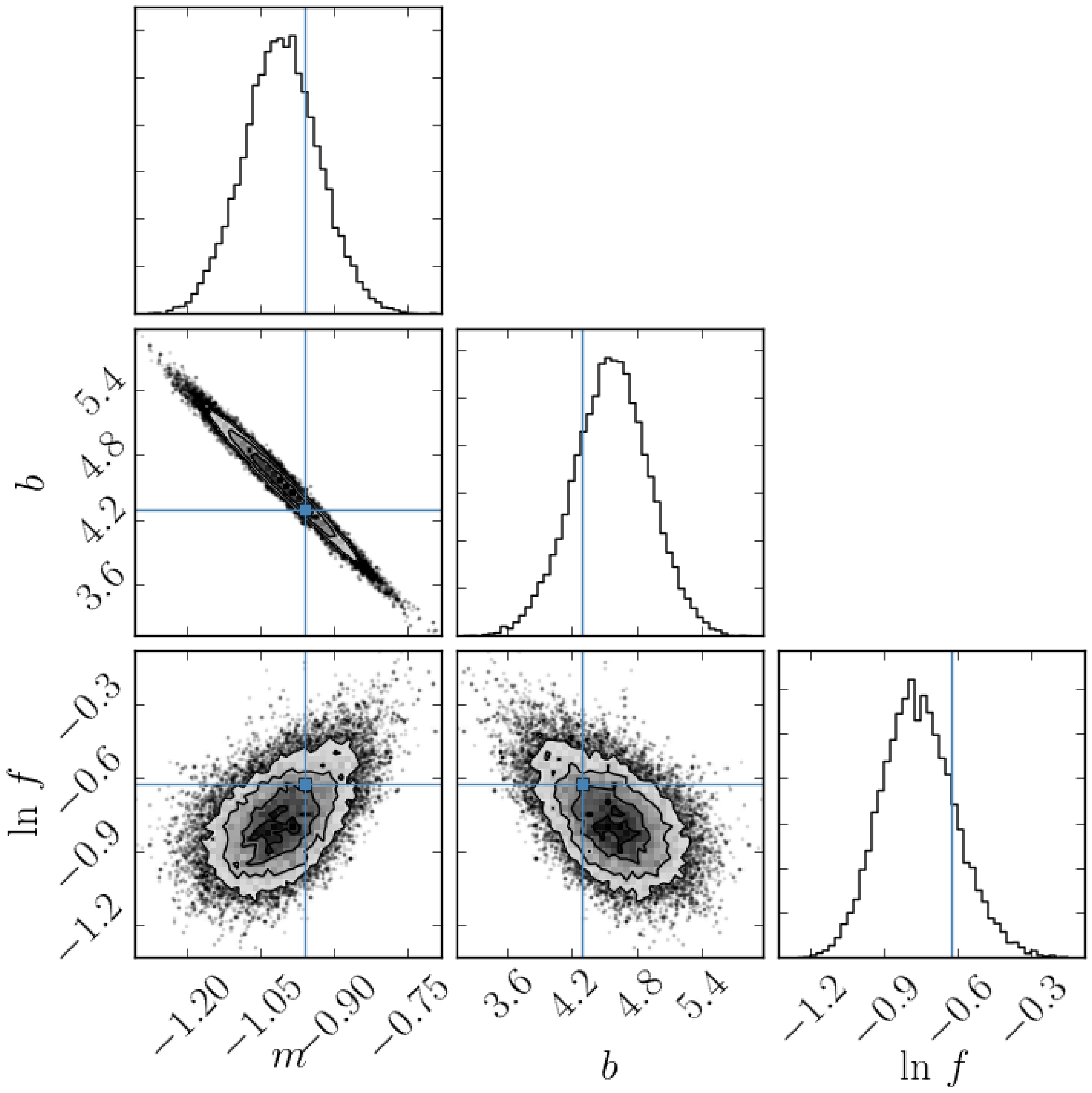
```
samples = sampler.chain[:, 50:, :].reshape((-1, ndim))
```

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:

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

and you should get something like the following:

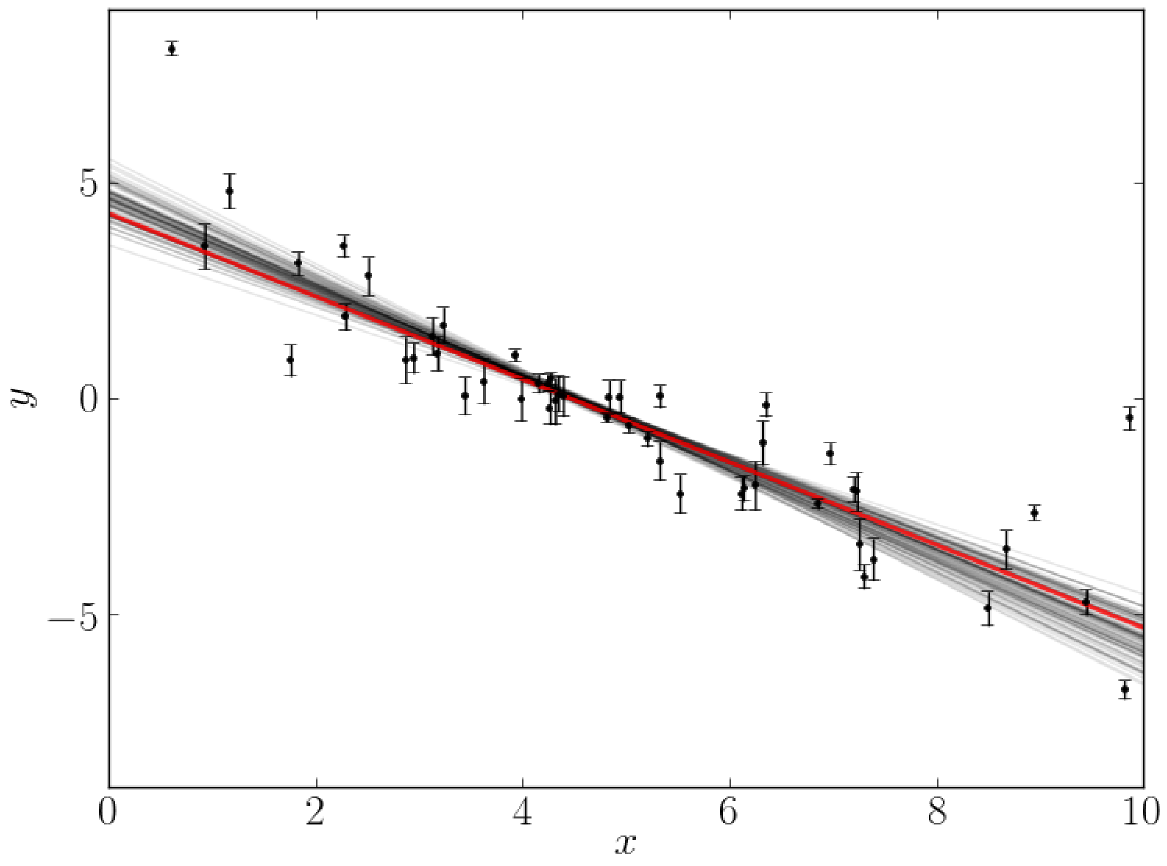


The corner plot shows all the one and two dimensional projections of the posterior probability distributions of your parameters. This is useful because it quickly demonstrates all of the covariances between parameters. Also, the way that you find the marginalized distribution for a parameter or set of 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:



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:

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

Advanced Patterns

emcee is generally pretty simple but it has a few key features that make the usage easier in real problems. Here are a few examples of things that you might find useful.

Incrementally saving progress

It is often useful to incrementally save the state of the chain to a file. This makes it easier to monitor the chain's progress and it makes things a little less disastrous if your code/computer crashes somewhere in the middle of an expensive MCMC run. If you just want to append the walker positions to the end of a file, you could do something like:

```
f = open("chain.dat", "w")
f.close()

for result in sampler.sample(pos0, iterations=500, storechain=False):
    position = result[0]
    f = open("chain.dat", "a")
    for k in range(position.shape[0]):
        f.write("{0:4d} {1:s}\n".format(k, " ".join(position[k])))
    f.close()
```

Printing the sampler's progress

You might want to monitor the progress of the sampler in your terminal while it runs. There are several modules out there that can help you make shiny progress bars (e.g., are progressbar and clint), but it's straightforward to implement a simple progress counter yourself.

The solution here is very similar to the incremental saving snippet. For example, to display the current percentage:

```
nsteps = 5000
for i, result in enumerate(sampler.sample(p0, iterations=nsteps)):
    if (i+1) % 100 == 0:
        print("{0:5.1%}".format(float(i) / nsteps))
```

Or, to display a rudimentary progress bar that updates iteself on a single line:

```
import sys

nsteps = 5000
width = 30
for i, result in enumerate(sampler.sample(p0, iterations=nsteps)):
    n = int((width+1) * float(i) / nsteps)
    sys.stdout.write("\r[{0}{1}]".format('#' * n, ' ' * (width - n)))
sys.stdout.write("\n")
```

Multiprocessing

In principle, running `emcee` in parallel is as simple instantiating an *EnsembleSampler* object with the `threads` argument set to an integer greater than 1:

```
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnpostfn, threads=15)
```

In practice, the parallelization is implemented using the built in Python multiprocessing module. With this comes a few constraints. In particular, both `lnpostfn` and `args` must be pickleable. The exceptions thrown while using `multiprocessing` can be quite cryptic and even though we've tried to make this feature as user-friendly as possible, it can sometimes cause some headaches. One useful debugging tactic is to try running with 1 thread if your processes start to crash. This will generally provide much more illuminating error messages than in the parallel case. Note that the parallelized *EnsembleSampler* object is not pickleable. Therefore, if it (or an object that contains it) is passed to `lnpostfn` when multiprocessing is turned on, the code will fail.

It is also important to note that the `multiprocessing` module works by spawning a large number of new `python` processes and running the code in isolation within those processes. This means that there is a significant amount of overhead involved at each step of the parallelization process. With this in mind, it is not surprising that running a simple problem like the *quickstart example* in parallel will run much slower than the equivalent serial code. If your log-probability function takes a significant amount of time (> 1 second or so) to compute then using the parallel sampler actually provides significant speed gains.

Arbitrary metadata blobs

Added in version 1.1.0

Imagine that your log-probability function involves an extremely computationally expensive numerical simulation starting from initial conditions parameterized by the position of the walker in parameter space. Then you have to compare the results of your simulation by projecting into data space (predicting you data) and computing something like a chi-squared scalar in this space. After you run MCMC, you might want to visualize the draws from your probability function in data space by over-plotting samples on your data points. It is obviously unreasonable to recompute all the simulations for all the initial conditions that you want to display as a part of your post-processing—especially since you already computed all of them before! Instead, it would be ideal to be able to store realizations associated with each step in the MCMC and then just display those after the fact. This is possible using the "arbitrary blob" pattern.

To use `blobs`, you just need to modify your log-probability function to return a second argument (this can be any arbitrary Python object). Then, the sampler object will have an attribute (called *EnsembleSampler.blobs*) that is a list (of length `niterations`) of lists (of length `nwalkers`) containing all the accepted `blobs` associated with the walker positions in *EnsembleSampler.chain*.

As an absolutely trivial example, let's say that we wanted to store the sum of cubes of the input parameters as a string at each position in the chain. To do this we could simply sample a function like:

```
def lnprobfn(p):
    return -0.5 * np.sum(p ** 2), str(np.sum(p ** 3))
```

It is important to note that by returning two values from our log-probability function, we also change the output of *EnsembleSampler.sample()* and *EnsembleSampler.run_mcmc()* to return 4 values (position, probability, random number generator state and blobs) instead of just the first three.

Using MPI to distribute the computations

Added in version 1.2.0

The standard implementation of `emcee` relies on the `multiprocessing` module to parallelize tasks. This works well on a single machine with multiple cores but it is sometimes useful to distribute the computation across a larger cluster. To do this, we need to do something a little bit more sophisticated using the mpi4py module. Below, we'll implement an example similar to the quickstart using MPI but first you'll need to install mpi4py.

The `utils.MPIPool` object provides most of the needed functionality so we'll start by importing that and the other needed modules:

```
import sys
import numpy as np
import emcee
from emcee.utils import MPIPool
```

This time, we'll just sample a simple isotropic Gaussian (remember that the `emcee` algorithm *doesn't care about covariances between parameters because it is affine-invariant*):

```
ndim = 50
nwalkers = 250
p0 = [np.random.rand(ndim) for i in xrange(nwalkers)]

def lnprob(x):
    return -0.5 * np.sum(x ** 2)
```

Now, this is where things start to change:

```
pool = MPIPool()
if not pool.is_master():
    pool.wait()
    sys.exit(0)
```

First, we're initializing the pool object and then—if the process isn't running as master—we wait for instructions and then exit. Then, we can set up the sampler providing this pool object to do the parallelization:

```
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob, pool=pool)
```

and then run and analyse as usual. The key here is that only the master chain should *actually* directly interact with the sampler and the other processes should only wait for instructions.

Note: don't forget to close the pool if you don't want the processes to hang forever:

```
pool.close()
```

The full source code for this example is available on Github.

If we save this script to the file `mpi.py`, we can then run this example with the command:

```
mpirun -np 2 python mpi.py
```

for local testing.

Loadbalancing in parallel runs

Added in version 2.1.0

When `emcee` is being used in a multi-processing mode (`multiprocessing` or `mpi4py`), the parameters need to distributed evenly over all the available cores. `emcee` uses a `map` function to distribute the jobs over the available cores. In case of `multiprocessing`, the `map` function is in-built and dynamically schedules the tasks. In order to get a similar dynamic scheduling in `map` when using *`utils.MPIPool`* , use the following invocation:

```
pool = MPIPool(loadbalance=True)
```

By default, `loadbalance` is set to `False`. If your jobs have a lot of variance in run-time, then setting the `loadbalance` option will improve the overall run-time.

If your problem is such that the runtime for each invocation of the log-probability function scales with one/some of the parameters, then you can improve load-balancing even further. By sorting the jobs in decreasing order of (expected) run-time, the longest jobs get run simultaneously and you only have the wait for the duration of the longest job. In the following example, the first parameter strongly determines the run-time – larger the first parameter, the longer the runtime. The `sort_on_runtime` returns the re-ordered list and the corresponding index.

```
def sort_on_runtime(pos):
    p = np.atleast_2d(pos)
    idx = np.argsort(p[:, 0])[::-1]
    return p[idx], idx
```

In order to use this function, you will have to instantiate an *`EnsembleSampler`* object with:

```
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob, pool=pool,
                                runtime_sortingfn=sort_on_runtime)
```

Such a `sort_on_runtime` can be applied to both `multiprocessing` and `mpi4py` invocations for `emcee`. You can see a benchmarking routine using the `mpi4py` module on Github.

Parallel-Tempering Ensemble MCMC

Added in version 1.2.0

When your posterior is multi-modal or otherwise hard to sample with a standard MCMC, a good option to try is parallel-tempered MCMC (PTMCMC). PTMCMC runs multiple MCMC's at different temperatures, T. Each MCMC samples from a modified posterior, given by

$$\pi_T(x) = [l(x)]^{\frac{1}{T}}\, p(x)$$

As $T \to \infty$, the posterior becomes the prior, which is hopefully easy to sample. If the likelihood is a Gaussian with standard deviation σ, then the tempered likelihood is proportional to a Gaussian with standard deviation $\sigma\sqrt{T}$.

Periodically during the run, the different temperatures swap members of their ensemble in a way that preserves detailed balance. The hot chains can more easily explore parameter space because the likelihood is flatter and broader, while the cold chains do a good job of exploring the peaks of the likelihood. This can **dramatically** improve convergence if your likelihood function has many well-separated modes.

How To Sample a Multi-Modal Gaussian

Suppose we want to sample from the posterior given by

$$\pi(\vec{x}) \propto \exp\left[-\frac{1}{2}\left(\vec{x}-\vec{\mu}_1\right)^T \Sigma_1^{-1}\left(\vec{x}-\vec{\mu}_1\right)\right] + \exp\left[-\frac{1}{2}\left(\vec{x}-\vec{\mu}_2\right)^T \Sigma_2^{-1}\left(\vec{x}-\vec{\mu}_2\right)\right]$$

If the modes $\mu_{1,2}$ are well-separated with respect to the scale of $\Sigma_{1,2}$, then this distribution will be hard to sample with the *`EnsembleSampler`*. Here is how we would sample from it using the *`PTSampler`*.

First, some preliminaries:

```python
import numpy as np
from emcee import PTSampler
```

Define the means and standard deviations of our multi-modal likelihood:

```python
# mu1 = [1, 1], mu2 = [-1, -1]
mu1 = np.ones(2)
mu2 = -np.ones(2)

# Width of 0.1 in each dimension
sigma1inv = np.diag([100.0, 100.0])
sigma2inv = np.diag([100.0, 100.0])

def logl(x):
    dx1 = x - mu1
    dx2 = x - mu2

    return np.logaddexp(-np.dot(dx1, np.dot(sigma1inv, dx1))/2.0,
                        -np.dot(dx2, np.dot(sigma2inv, dx2))/2.0)

# Use a flat prior
def logp(x):
    return 0.0
```

Now we can construct a sampler object that will drive the PTMCMC; arbitrarily, we choose to use 20 temperatures (the default is for each temperature to increase by a factor of $\sqrt{2}$, so the highest temperature will be $T = 1024$, resulting in an effective $\sigma_T = 32\sigma = 3.2$, which is about the separation of our modes). Let's use 100 walkers in the ensemble at each temperature:

```python
ntemps = 20
nwalkers = 100
ndim = 2

sampler=PTSampler(ntemps, nwalkers, ndim, logl, logp)
```

Making the sampling multi-threaded is as simple as adding the `threads=Nthreads` argument to *`PTSampler`*. We could have modified the temperature ladder using the `betas` optional argument (which should be an array of $\beta \equiv 1/T$ values). The `pool` argument also allows to specify our own pool of worker threads if we wanted fine-grained control over the parallelism.

First, we run the sampler for 1000 burn-in iterations:

```python
p0 = np.random.uniform(low=-1.0, high=1.0, size=(ntemps, nwalkers, ndim))
for p, lnprob, lnlike in sampler.sample(p0, iterations=1000):
    pass
sampler.reset()
```

Now we sample for 10000 iterations, recording every 10th sample:

```python
for p, lnprob, lnlike in sampler.sample(p, lnprob0=lnprob,
                                        lnlike0=lnlike,
                                        iterations=10000, thin=10):
    pass
```

The resulting samples (1000 of them) are stored as the `sampler.chain` property:

```
assert sampler.chain.shape == (ntemps, nwalkers, 1000, ndim)

# Chain has shape (ntemps, nwalkers, nsteps, ndim)
# Zero temperature mean:
mu0 = np.mean(np.mean(sampler.chain[0,...], axis=0), axis=0)

# Longest autocorrelation length (over any temperature)
max_acl = np.max(sampler.acor)

# etc
```

Implementation Notes

For a description of the parallel-tempering algorithm, see, e.g. Earl & Deem (2010), Phys Chem Chem Phys, 7, 23, 3910. The algorithm has some tunable parameters:

Temperature Ladder The choice of temperature for the chains will strongly influence the rate of convergence of the sampling. By default, the *PTSampler* class uses an exponential ladder, with each temperature increasing by a factor of $\sqrt{2}$. The user can supply their own ladder using the `beta` optional argument in the constructor.

Rate of Temperature Swaps The rate at which temperature swaps are proposed can, to a lesser extent, also influence the rate of convergence of the sampling. The goal is to make sure that good positions found by the high temperatures can propogate to the lower temperatures, but still ensure that the high-temperatures do not lose all memory of good locations. Here we choose to implement one temperature swap proposal per walker per rung on the temperature ladder after each ensemble update. This is not user-tunable, but seems to work well in practice.

The `args` optional argument is not available in the *PTSampler* constructor; use a custom class with a `__call__` method if you need to pass arguments to the `lnlike` or `lnprior` functions and do not want to use a global variable.

The `thermodynamic_integration_log_evidence` uses thermodynamic integration (see, e.g., Goggans & Chi (2004), AIP Conf Proc, 707, 59) to estimate the evidence integral. Define the evidence as a function of inverse temperature:

$$Z(\beta) \equiv \int dx\, l^\beta(x) p(x)$$

We want to compute $Z(1)$. Z satisfies the following differential equation

$$\frac{d \ln Z}{d\beta} = \frac{1}{Z(\beta)} \int dx\, \ln l(x) l^\beta(x) p(x) = \langle \ln l \rangle_\beta$$

where $\langle \ldots \rangle_\beta$ is the average of a quantity over the posterior at temperature $T = 1/\beta$. Integrating (note that $Z(0) = 1$ because the prior is normalized), we have

$$\ln Z(1) = \int_0^1 d\beta\, \langle \ln l \rangle_\beta$$

This quantity can be estimated from a PTMCMC by computing the average $\ln l$ within each chain and applying a quadrature formula to estimate the integral.

FAQ

The not-so-frequently asked questions that still have useful answers

What are "walkers"?

Walkers are the members of the ensemble. They are almost like separate Metropolis-Hastings chains but, of course, the proposal distribution for a given walker depends on the positions of all the other walkers in the ensemble. See Goodman & Weare (2010) for more details.

How should I initialize the walkers?

The best technique seems to be to start in a small ball around the a priori preferred position. Don't worry, the walkers quickly branch out and explore the rest of the space.

Wrapping C++ code

There are numerous ways to do it, see the python wiki.

Extra care has to be taken if mpi support is needed as the mpi4py module used by emcee depends on the pickle module to send a function call to different processors/cores.

A minimal extension of the mpi.py example in which the target density is coded in C++ and wrapped with the swig library is shown in this gist. It also demonstrates the hacks needed to get the pickling to work.

Parameter limits

In order to confine the walkers to a finite volume of the parameter space, have your function return negative infinity outside of the volume corresponding to the logarithm of 0 prior probability using:

```
return -numpy.inf
```

Note: if your function is written in C++, use:

```
return -std::numeric_limits<double>::infinity();
```

and avoid:

```
return -std::numeric_limits<double>::max();
```

as it does not have the desired effect.

Troubleshooting

I'm getting weird spikes in my data/I have low acceptance fractions/both... what should I do?

Double the number of walkers. If that doesn't work, double it again. And again. Until you run out of RAM. At that point, I don't know!

The walkers are getting stuck in "islands" of low likelihood. How can I fix that?

Try increasing the number of walkers. If that doesn't work, you can try pruning using a clustering algorithm like the one found in arxiv:1104.2612.

CHAPTER 4

API Documentation

API

This page details the methods and classes provided by the `emcee` module. The main entry point is through the *`EnsembleSampler`* object.

The Affine-Invariant Ensemble Sampler

Standard usage of `emcee` involves instantiating an *`EnsembleSampler`*.

class `emcee.`**`EnsembleSampler`**(*nwalkers*, *dim*, *lnpostfn*, *a=2.0*, *args=[]*, *kwargs={}*, *postargs=None*, *threads=1*, *pool=None*, *live_dangerously=False*, *runtime_sortingfn=None*)

A generalized Ensemble sampler that uses 2 ensembles for parallelization. The `__init__` function will raise an `AssertionError` if `nwalkers < 2 * dim` (and you haven't set the `live_dangerously` parameter) or if `nwalkers` is odd.

Warning: The *`chain`* member of this object has the shape: `(nwalkers, nlinks, dim)` where `nlinks` is the number of steps taken by the chain and `nwalkers` is the number of walkers. Use the *`flatchain`* property to get the chain flattened to `(nlinks, dim)`. For users of pre-1.0 versions, this shape is different so be careful!

Parameters

- **`nwalkers`** – The number of Goodman & Weare "walkers".
- **`dim`** – Number of dimensions in the parameter space.
- **`lnpostfn`** – A function that takes a vector in the parameter space as input and returns the natural logarithm of the posterior probability for that position.
- **`a`** – (optional) The proposal scale parameter. (default: `2.0`)
- **`args`** – (optional) A list of extra positional arguments for `lnpostfn`. `lnpostfn` will be called with the sequence `lnpostfn(p, *args, **kwargs)`.

- **kwargs** – (optional) A list of extra keyword arguments for `lnpostfn`. `lnpostfn` will be called with the sequence `lnpostfn(p, *args, **kwargs)`.
- **postargs** – (optional) Alias of `args` for backwards compatibility.
- **threads** – (optional) The number of threads to use for parallelization. If `threads == 1`, then the `multiprocessing` module is not used but if `threads > 1`, then a `Pool` object is created and calls to `lnpostfn` are run in parallel.
- **pool** – (optional) An alternative method of using the parallelized algorithm. If provided, the value of `threads` is ignored and the object provided by `pool` is used for all parallelization. It can be any object with a `map` method that follows the same calling sequence as the built-in `map` function.
- **runtime_sortingfn** – (optional) A function implementing custom runtime load-balancing. See *Loadbalancing in parallel runs* for more information.

acceptance_fraction
An array (length: `k`) of the fraction of steps accepted for each walker.

acor
An estimate of the autocorrelation time for each parameter (length: `dim`).

blobs
Get the list of "blobs" produced by sampling. The result is a list (of length `iterations`) of `list` s (of length `nwalkers`) of arbitrary objects. **Note**: this will actually be an empty list if your `lnpostfn` doesn't return any metadata.

chain
A pointer to the Markov chain itself. The shape of this array is `(k, iterations, dim)`.

clear_blobs()
Clear the `blobs` list.

clear_chain()
An alias for `reset()` kept for backwards compatibility.

flatchain
A shortcut for accessing chain flattened along the zeroth (walker) axis.

flatlnprobability
A shortcut to return the equivalent of `lnprobability` but aligned to `flatchain` rather than `chain`. The shape is `(k * iterations)`.

get_autocorr_time(*low=10*, *high=None*, *step=1*, *c=10*, *fast=False*)
Compute an estimate of the autocorrelation time for each parameter (length: `dim`).

Parameters

- **low** – (Optional[int]) The minimum window size to test. (default: `10`)
- **high** – (Optional[int]) The maximum window size to test. (default: `x.shape[axis] / (2*c)`)
- **step** – (Optional[int]) The step size for the window search. (default: `1`)
- **c** – (Optional[float]) The minimum number of autocorrelation times needed to trust the estimate. (default: `10`)
- **fast** – (Optional[bool]) If `True`, only use the first `2^n` (for the largest power) entries for efficiency. (default: False)

get_lnprob(*p*)
Return the log-probability at the given position.

lnprobability
A pointer to the matrix of the value of `lnprobfn` produced at each step for each walker. The shape is `(k, iterations)`.

random_state
The state of the internal random number generator. In practice, it's the result of calling `get_state()` on a `numpy.random.mtrand.RandomState` object. You can try to set this property but be warned that if you do this and it fails, it will do so silently.

reset()
Clear the `chain` and `lnprobability` array. Also reset the bookkeeping parameters.

run_mcmc(*pos0*, *N*, *rstate0=None*, *lnprob0=None*, ***kwargs*)
Iterate `sample()` for `N` iterations and return the result.

Parameters

- **pos0** – The initial position vector. Can also be None to resume from where :func:`run_mcmc` left off the last time it executed.
- **N** – The number of steps to run.
- **lnprob0** – (optional) The log posterior probability at position `p0`. If `lnprob` is not provided, the initial value is calculated.
- **rstate0** – (optional) The state of the random number generator. See the `random_state()` property for details.
- **kwargs** – (optional) Other parameters that are directly passed to `sample()`.

This method returns the most recent result from `sample()`. The particular values vary from sampler to sampler, but the result is generally a tuple `(pos, lnprob, rstate)` or `(pos, lnprob, rstate, blobs)` where `pos` is the most recent position vector (or ensemble thereof), `lnprob` is the most recent log posterior probability (or ensemble thereof), `rstate` is the state of the random number generator, and the optional `blobs` are user-provided large data blobs.

sample(*p0*, *lnprob0=None*, *rstate0=None*, *blobs0=None*, *iterations=1*, *thin=1*, *storechain=True*, *mh_proposal=None*)
Advance the chain `iterations` steps as a generator.

Parameters

- **p0** – A list of the initial positions of the walkers in the parameter space. It should have the shape `(nwalkers, dim)`.
- **lnprob0** – (optional) The list of log posterior probabilities for the walkers at positions given by `p0`. If `lnprob is None`, the initial values are calculated. It should have the shape `(k, dim)`.
- **rstate0** – (optional) The state of the random number generator. See the `Sampler.random_state` property for details.
- **iterations** – (optional) The number of steps to run.
- **thin** – (optional) If you only want to store and yield every `thin` samples in the chain, set thin to an integer greater than 1.
- **storechain** – (optional) By default, the sampler stores (in memory) the positions and log-probabilities of the samples in the chain. If you are using another method to store the samples to a file or if you don't need to analyse the samples after the fact (for burn-in for example) set `storechain` to `False`.

- **mh_proposal** – (optional) A function that returns a list of positions for `nwalkers` walkers given a current list of positions of the same size. See *`utils.MH_proposal_axisaligned`* for an example.

At each iteration, this generator yields:

- `pos` - A list of the current positions of the walkers in the parameter space. The shape of this object will be `(nwalkers, dim)`.
- `lnprob` - The list of log posterior probabilities for the walkers at positions given by `pos` . The shape of this object is `(nwalkers,)`.
- `rstate` - The current state of the random number generator.
- `blobs` - (optional) The metadata "blobs" associated with the current position. The value is only returned if `lnpostfn` returns blobs too.

The Parallel-Tempered Ensemble Sampler

The *`PTSampler`* class performs a parallel-tempered ensemble sampling using *`EnsembleSampler`* to sample within each temperature. This sort of sampling is useful if you expect your distribution to be multi-modal. Take a look at *the documentation* to see how you might use this class.

class `emcee.`**`PTSampler`**(*ntemps*, *nwalkers*, *dim*, *logl*, *logp*, *threads=1*, *pool=None*, *betas=None*, *a=2.0*, *Tmax=None*, *loglargs=[]*, *logpargs=[]*, *loglkwargs={}*, *logpkwargs={}*)

A parallel-tempered ensemble sampler, using *`EnsembleSampler`* for sampling within each parallel chain.

Parameters

- **ntemps** – The number of temperatures. Can be `None`, in which case the `Tmax` argument sets the maximum temperature.
- **nwalkers** – The number of ensemble walkers at each temperature.
- **dim** – The dimension of parameter space.
- **logl** – The log-likelihood function.
- **logp** – The log-prior function.
- **threads** – (optional) The number of parallel threads to use in sampling.
- **pool** – (optional) Alternative to `threads`. Any object that implements a `map` method compatible with the built-in `map` will do here. For example, `multi.Pool` will do.
- **betas** – (optional) Array giving the inverse temperatures, $\beta = 1/T$, used in the ladder. The default is chosen so that a Gaussian posterior in the given number of dimensions will have a 0.25 tswap acceptance rate.
- **a** – (optional) Proposal scale factor.
- **Tmax** – (optional) Maximum temperature for the ladder. If `ntemps` is `None`, this argument is used to set the temperature ladder.
- **loglargs** – (optional) Positional arguments for the log-likelihood function.
- **logpargs** – (optional) Positional arguments for the log-prior function.
- **loglkwargs** – (optional) Keyword arguments for the log-likelihood function.
- **logpkwargs** – (optional) Keyword arguments for the log-prior function.

`acceptance_fraction`

Matrix of shape `(Ntemps, Nwalkers)` detailing the acceptance fraction for each walker.

acor
Returns a matrix of autocorrelation lengths for each parameter in each temperature of shape `(Ntemps, Ndim)`.

betas
Returns the sequence of inverse temperatures in the ladder.

chain
Returns the stored chain of samples; shape `(Ntemps, Nwalkers, Nsteps, Ndim)`.

clear_chain()
An alias for *`reset()`* kept for backwards compatibility.

flatchain
Returns the stored chain, but flattened along the walker axis, so of shape `(Ntemps, Nwalkers*Nsteps, Ndim)`.

get_autocorr_time(***kwargs*)
Returns a matrix of autocorrelation lengths for each parameter in each temperature of shape `(Ntemps, Ndim)`.

Any arguments will be passed to `autocorr.integrate_time()`.

get_lnprob(*p*)
Return the log-probability at the given position.

lnlikelihood
Matrix of ln-likelihood values; shape `(Ntemps, Nwalkers, Nsteps)`.

lnprobability
Matrix of lnprobability values; shape `(Ntemps, Nwalkers, Nsteps)`.

random_state
The state of the internal random number generator. In practice, it's the result of calling `get_state()` on a `numpy.random.mtrand.RandomState` object. You can try to set this property but be warned that if you do this and it fails, it will do so silently.

reset()
Clear the `chain`, `lnprobability`, `lnlikelihood`, `acceptance_fraction`, `tswap_acceptance_fraction` stored properties.

run_mcmc(*pos0*, *N*, *rstate0=None*, *lnprob0=None*, ***kwargs*)
Iterate *`sample()`* for `N` iterations and return the result.

Parameters

- **pos0** – The initial position vector. Can also be None to resume from where :func:`run_mcmc` left off the last time it executed.
- **N** – The number of steps to run.
- **lnprob0** – (optional) The log posterior probability at position `p0`. If `lnprob` is not provided, the initial value is calculated.
- **rstate0** – (optional) The state of the random number generator. See the *`random_state()`* property for details.
- **kwargs** – (optional) Other parameters that are directly passed to *`sample()`*.

This method returns the most recent result from *`sample()`*. The particular values vary from sampler to sampler, but the result is generally a tuple `(pos, lnprob, rstate)` or `(pos, lnprob, rstate, blobs)` where `pos` is the most recent position vector (or ensemble thereof), `lnprob` is the most recent log posterior probability (or ensemble thereof), `rstate` is the state of the random number generator, and the optional `blobs` are user-provided large data blobs.

sample (*p0*, *lnprob0=None*, *lnlike0=None*, *rstate0=None*, *iterations=1*, *thin=1*, *storechain=True*)

Advance the chains `iterations` steps as a generator.

Parameters

- **p0** – The initial positions of the walkers. Shape should be `(ntemps, nwalkers, dim)`.
- **lnprob0** – (optional) The initial posterior values for the ensembles. Shape `(ntemps, nwalkers)`.
- **lnlike0** – (optional) The initial likelihood values for the ensembles. Shape `(ntemps, nwalkers)`.
- **rstate0** – (optional) The state of the random number generator. See the `Sampler.random_state` property for details.
- **iterations** – (optional) The number of iterations to preform.
- **thin** – (optional) The number of iterations to perform between saving the state to the internal chain.
- **storechain** – (optional) If `True` store the iterations in the `chain` property.

At each iteration, this generator yields

- `p`, the current position of the walkers.
- `lnprob` the current posterior values for the walkers.
- `lnlike` the current likelihood values for the walkers.

thermodynamic_integration_log_evidence (*logls=None*, *fburnin=0.1*)

Thermodynamic integration estimate of the evidence.

Parameters

- **logls** – (optional) The log-likelihoods to use for computing the thermodynamic evidence. If `None` (the default), use the stored log-likelihoods in the sampler. Should be of shape `(Ntemps, Nwalkers, Nsamples)`.
- **fburnin** – (optional) The fraction of the chain to discard as burnin samples; only the final `1-fburnin` fraction of the samples will be used to compute the evidence; the default is `fburnin = 0.1`.

Return (lnZ, dlnZ) `(lnZ, dlnZ)` Returns an estimate of the log-evidence and the error associated with the finite number of temperatures at which the posterior has been sampled.

The evidence is the integral of the un-normalized posterior over all of parameter space:

$$Z \equiv \int d\theta \, l(\theta) p(\theta)$$

Thermodymanic integration is a technique for estimating the evidence integral using information from the chains at various temperatures. Let

$$Z(\beta) = \int d\theta \, l^\beta(\theta) p(\theta)$$

Then

$$\frac{d \ln Z}{d\beta} = \frac{1}{Z(\beta)} \int d\theta l^\beta p \ln l = \left\langle \ln l \right\rangle_\beta$$

so

$$\ln Z(\beta = 1) = \int_0^1 d\beta \, \langle \ln l \rangle_\beta$$

By computing the average of the log-likelihood at the difference temperatures, the sampler can approximate the above integral.

tswap_acceptance_fraction
Returns an array of accepted temperature swap fractions for each temperature; shape `(ntemps, )`.

Standard Metropolis-Hastings Sampler

The Metropolis-Hastings sampler included in this module is far from fine-tuned and optimized. It is, however, stable and it has a consistent API so it can be useful for testing and comparison.

class `emcee.`**`MHSampler`**(*cov*, **args*, ***kwargs*)
The most basic possible Metropolis-Hastings style MCMC sampler

Parameters

- **cov** – The covariance matrix to use for the proposal distribution.
- **dim** – Number of dimensions in the parameter space.
- **lnpostfn** – A function that takes a vector in the parameter space as input and returns the natural logarithm of the posterior probability for that position.
- **args** – (optional) A list of extra positional arguments for `lnpostfn`. `lnpostfn` will be called with the sequence `lnpostfn(p, *args, **kwargs)`.
- **kwargs** – (optional) A list of extra keyword arguments for `lnpostfn`. `lnpostfn` will be called with the sequence `lnpostfn(p, *args, **kwargs)`.

acceptance_fraction
The fraction of proposed steps that were accepted.

acor
An estimate of the autocorrelation time for each parameter (length: `dim`).

chain
A pointer to the Markov chain.

clear_chain()
An alias for `reset()` kept for backwards compatibility.

flatchain
Alias of `chain` provided for compatibility.

get_autocorr_time(*low=10*, *high=None*, *step=1*, *c=10*, *fast=False*)
Compute an estimate of the autocorrelation time for each parameter (length: `dim`).

Parameters

- **low** – (Optional[int]) The minimum window size to test. (default: `10`)
- **high** – (Optional[int]) The maximum window size to test. (default: `x.shape[axis] / (2*c)`)
- **step** – (Optional[int]) The step size for the window search. (default: `1`)
- **c** – (Optional[float]) The minimum number of autocorrelation times needed to trust the estimate. (default: `10`)

- **fast** – (Optional[bool]) If `True`, only use the first `2^n` (for the largest power) entries for efficiency. (default: False)

get_lnprob(*p*)
Return the log-probability at the given position.

lnprobability
A list of the log-probability values associated with each step in the chain.

random_state
The state of the internal random number generator. In practice, it's the result of calling `get_state()` on a `numpy.random.mtrand.RandomState` object. You can try to set this property but be warned that if you do this and it fails, it will do so silently.

run_mcmc(*pos0*, *N*, *rstate0=None*, *lnprob0=None*, ***kwargs*)
Iterate *`sample()`* for `N` iterations and return the result.

Parameters

- **pos0** – The initial position vector. Can also be None to resume from where :func:`run_mcmc` left off the last time it executed.
- **N** – The number of steps to run.
- **lnprob0** – (optional) The log posterior probability at position `p0`. If `lnprob` is not provided, the initial value is calculated.
- **rstate0** – (optional) The state of the random number generator. See the *`random_state()`* property for details.
- **kwargs** – (optional) Other parameters that are directly passed to *`sample()`*.

This method returns the most recent result from *`sample()`*. The particular values vary from sampler to sampler, but the result is generally a tuple `(pos, lnprob, rstate)` or `(pos, lnprob, rstate, blobs)` where `pos` is the most recent position vector (or ensemble thereof), `lnprob` is the most recent log posterior probability (or ensemble thereof), `rstate` is the state of the random number generator, and the optional `blobs` are user-provided large data blobs.

sample(*p0*, *lnprob=None*, *rstate0=None*, *thin=1*, *storechain=True*, *iterations=1*)
Advances the chain `iterations` steps as an iterator

Parameters

- **p0** – The initial position vector.
- **lnprob0** – (optional) The log posterior probability at position `p0`. If `lnprob` is not provided, the initial value is calculated.
- **rstate0** – (optional) The state of the random number generator. See the *`random_state()`* property for details.
- **iterations** – (optional) The number of steps to run.
- **thin** – (optional) If you only want to store and yield every `thin` samples in the chain, set thin to an integer greater than 1.
- **storechain** – (optional) By default, the sampler stores (in memory) the positions and log-probabilities of the samples in the chain. If you are using another method to store the samples to a file or if you don't need to analyse the samples after the fact (for burn-in for example) set `storechain` to `False`.

At each iteration, this generator yields:

- `pos` - The current positions of the chain in the parameter space.

- `lnprob` - The value of the log posterior at `pos` .
- `rstate` - The current state of the random number generator.

Abstract Sampler Object

This section is mostly for developers who would be interested in implementing a new sampler for inclusion in `emcee`. A good starting point would be to subclass the sampler object and override the `Sampler.sample()` method.

class `emcee.`**`Sampler`**(*dim*, *lnprobfn*, *args=[]*, *kwargs={}*)

An abstract sampler object that implements various helper functions

Parameters

- **`dim`** – The number of dimensions in the parameter space.
- **`lnpostfn`** – A function that takes a vector in the parameter space as input and returns the natural logarithm of the posterior probability for that position.
- **`args`** – (optional) A list of extra positional arguments for `lnpostfn`. `lnpostfn` will be called with the sequence `lnpostfn(p, *args, **kwargs)`.
- **`kwargs`** – (optional) A list of extra keyword arguments for `lnpostfn`. `lnpostfn` will be called with the sequence `lnpostfn(p, *args, **kwargs)`.

`acceptance_fraction`

The fraction of proposed steps that were accepted.

`chain`

A pointer to the Markov chain.

`clear_chain``()`

An alias for `reset()` kept for backwards compatibility.

`flatchain`

Alias of `chain` provided for compatibility.

`get_lnprob`(*p*)

Return the log-probability at the given position.

`lnprobability`

A list of the log-probability values associated with each step in the chain.

`random_state`

The state of the internal random number generator. In practice, it's the result of calling `get_state()` on a `numpy.random.mtrand.RandomState` object. You can try to set this property but be warned that if you do this and it fails, it will do so silently.

`reset``()`

Clear `chain`, `lnprobability` and the bookkeeping parameters.

`run_mcmc`(*pos0*, *N*, *rstate0=None*, *lnprob0=None*, ***kwargs*)

Iterate `sample()` for `N` iterations and return the result.

Parameters

- **`pos0`** – The initial position vector. Can also be None to resume from where :func:`run_mcmc` left off the last time it executed.
- **`N`** – The number of steps to run.
- **`lnprob0`** – (optional) The log posterior probability at position `p0`. If `lnprob` is not provided, the initial value is calculated.

- **rstate0** – (optional) The state of the random number generator. See the *random_state()* property for details.
- **kwargs** – (optional) Other parameters that are directly passed to `sample()`.

This method returns the most recent result from `sample()`. The particular values vary from sampler to sampler, but the result is generally a tuple `(pos, lnprob, rstate)` or `(pos, lnprob, rstate, blobs)` where `pos` is the most recent position vector (or ensemble thereof), `lnprob` is the most recent log posterior probability (or ensemble thereof), `rstate` is the state of the random number generator, and the optional `blobs` are user-provided large data blobs.

Autocorrelation Analysis

A good heuristic for assessing convergence of samplings is the integrated autocorrelation time. `emcee` includes (as of version 2.1.0) tools for computing this and the autocorrelation function itself.

`emcee.autocorr.`**`integrated_time`**(*x*, *low=10*, *high=None*, *step=1*, *c=10*, *full_output=False*, *axis=0*, *fast=False*)

Estimate the integrated autocorrelation time of a time series.

This estimate uses the iterative procedure described on page 16 of Sokal's notes to determine a reasonable window size.

Args:

x: The time series. If multidimensional, set the time axis using the `axis` keyword argument and the function will be computed for every other axis.

low (Optional[int]): The minimum window size to test. (default: `10`) high (Optional[int]): The maximum window size to test. (default:

```
x.shape[axis] / (2*c))
```

step (Optional[int]): The step size for the window search. (default: `1`)

c (Optional[float]): The minimum number of autocorrelation times needed to trust the estimate. (default: `10`)

full_output (Optional[bool]): Return the final window size as well as the autocorrelation time. (default: `False`)

axis (Optional[int]): The time axis of `x`. Assumed to be the first axis if not specified.

fast (Optional[bool]): If `True`, only use the first `2^n` (for the largest power) entries for efficiency. (default: False)

Returns:

float or array: An estimate of the integrated autocorrelation time of the time series `x` computed along the axis `axis`.

Optional[int]: The final window size that was used. Only returned if `full_output` is `True`.

Raises

AutocorrError: If the autocorrelation time can't be reliably estimated from the chain. This normally means that the chain is too short.

`emcee.autocorr.`**`function`**(*x*, *axis=0*, *fast=False*)

Estimate the autocorrelation function of a time series using the FFT.

Args:

x: The time series. If multidimensional, set the time axis using the `axis` keyword argument and the function will be computed for every other axis.

axis (Optional[int]): The time axis of `x`. Assumed to be the first axis if not specified.

fast (Optional[bool]): If `True`, only use the first `2^n` (for the largest power) entries for efficiency. (default: False)

Returns: array: The autocorrelation function of the time series.

Utilities

`emcee.utils.`**`sample_ball`**(*p0*, *std*, *size=1*)

Produce a ball of walkers around an initial parameter value.

Parameters

- **`p0`** – The initial parameter value.
- **`std`** – The axis-aligned standard deviation.
- **`size`** – The number of samples to produce.

class `emcee.utils.`**`MH_proposal_axisaligned`**(*stdev*)

A Metropolis-Hastings proposal, with axis-aligned Gaussian steps, for convenient use as the `mh_proposal` option to *`EnsembleSampler.sample()`* .

Pools

These are some helper classes for using the built-in parallel version of the algorithm. These objects can be initialized and then passed into the constructor for the *`EnsembleSampler`* object using the `pool` keyword argument.

Interruptible Pool

Python's multiprocessing.Pool class doesn't interact well with `KeyboardInterrupt` signals, as documented in places such as:

- http://stackoverflow.com/questions/1408356/
- http://stackoverflow.com/questions/11312525/
- http://noswap.com/blog/python-multiprocessing-keyboardinterrupt

Various workarounds have been shared. Here, we adapt the one proposed in the last link above, by John Reese, and shared as

- https://github.com/jreese/multiprocessing-keyboardinterrupt/

Our version is a drop-in replacement for multiprocessing.Pool ... as long as the map() method is the only one that needs to be interrupt-friendly.

Contributed by Peter K. G. Williams <peter@newton.cx>.

Added in version 2.1.0

class `emcee.interruptible_pool.`**`InterruptiblePool`**(*processes=None*, *initializer=None*, *initargs=()*, ***kwargs*)

A modified version of `multiprocessing.pool.Pool` that has better behavior with regard to `KeyboardInterrupts` in the *`map()`* method.

Parameters

- **processes** – (optional) The number of worker processes to use; defaults to the number of CPUs.
- **initializer** – (optional) Either `None`, or a callable that will be invoked by each worker process when it starts.
- **initargs** – (optional) Arguments for *initializer*; it will be called as `initializer(*initargs)`.
- **kwargs** – (optional) Extra arguments. Python 2.7 supports a `maxtasksperchild` parameter.

map (*func*, *iterable*, *chunksize=None*)
Equivalent of `map()` built-in, without swallowing `KeyboardInterrupt`.

Parameters

- **func** – The function to apply to the items.
- **iterable** – An iterable of items that will have *func* applied to them.

MPI Pool

Built-in support for MPI distributed systems. See the documentation: *Using MPI to distribute the computations*.

class `emcee.utils.`**MPIPool** (*comm=None*, *debug=False*, *loadbalance=False*)
A pool that distributes tasks over a set of MPI processes. MPI is an API for distributed memory parallelism. This pool will let you run emcee without shared memory, letting you use much larger machines with emcee.

The pool only support the `map()` method at the moment because this is the only functionality that emcee needs. That being said, this pool is fairly general and it could be used for other purposes.

Contributed by Joe Zuntz.

Parameters

- **comm** – (optional) The `mpi4py` communicator.
- **debug** – (optional) If `True`, print out a lot of status updates at each step.
- **loadbalance** – (optional) if `True` and ntask > Ncpus, tries to loadbalance by sending out one task to each cpu first and then sending out the rest as the cpus get done.

bcast (**args*, ***kwargs*)
Equivalent to mpi4py `bcast()` collective operation.

close ()
Just send a message off to all the pool members which contains the special `_close_pool_message` sentinel.

is_master ()
Is the current process the master?

map (*function*, *tasks*)
Like the built-in `map()` function, apply a function to all of the values in a list and return the list of results.

Parameters

- **function** – The function to apply to the list.
- **tasks** – The list of elements.

wait ()
If this isn't the master process, wait for instructions.

CHAPTER 5

Contributors

Author:

- Dan Foreman-Mackey (NYU)

Direct contributions to the code base:

- Ruth Angus
- Bence Béky
- Frederik Beaujean
- Alex Conley
- Miguel de Val-Borro
- Will Meierjurgen Farr
- Júlio Hoffimann Mendes
- David W. Hogg
- Dustin Lang
- Phil Marshall
- Demitri Muna
- Adrian Price-Whelan
- Jeremy Sanders
- Leo Singer
- Manodeep Sinha
- Marco Tazzari
- Erik Tollerud
- Simon Walker
- Peter K. G. Williams

- Joe Zuntz

Comments, corrections & suggestions:

- Eric Agol
- Jo Bovy
- Andrew Bradshaw
- Jacqueline Chen
- John Gizis
- Jonathan Goodman
- Jennifer Piscionere

CHAPTER 6

License & Attribution

If you make use of emcee in your work, please cite our paper (arXiv, ADS, BibTeX) and consider adding your paper to the testimonials list.

CHAPTER 7

Changelog

2.2.0 (2016-07-12)

- Improved autocorrelation time computation.
- Numpy compatibility issues.
- Fixed deprecated integer division behavior in PTSampler.

2.1.0 (2014-05-22)

- Removing dependence on `acor` extension.
- Added arguments to `PTSampler` function.
- Added automatic load-balancing for MPI runs.
- Added custom load-balancing for MPI and multiprocessing.
- New default multiprocessing pool that supports `^C`.

2.0.0 (2013-11-17)

- **Re-licensed under the MIT license!**
- Clearer less verbose documentation.
- Added checks for parameters becoming infinite or NaN.
- Added checks for log-probability becoming NaN.
- Improved parallelization and various other tweaks in `PTSampler`.

1.2.0 (2013-01-30)

- Added a parallel tempering sampler `PTSampler`.
- Added instructions and utilities for using `emcee` with `MPI`.
- Added `flatlnprobability` property to the `EnsembleSampler` object to be consistent with the `flatchain` property.
- Updated document for publication in PASP.
- Various bug fixes.

1.1.3 (2012-11-22)

- Made the packaging system more robust even when numpy is not installed.

1.1.2 (2012-08-06)

- Another bug fix related to metadata blobs: the shape of the final `blobs` object was incorrect and all of the entries would generally be identical because we needed to copy the list that was appended at each step. Thanks goes to Jacqueline Chen (MIT) for catching this problem.

1.1.1 (2012-07-30)

- Fixed bug related to metadata blobs. The sample function was yielding the `blobs` object even when it wasn't expected.

1.1.0 (2012-07-28)

- Allow the `lnprobfn` to return arbitrary "blobs" of data as well as the log-probability.
- Python 3 compatible (thanks Alex Conley)!
- Various speed ups and clean ups in the core code base.
- New documentation with better examples and more discussion.

1.0.1 (2012-03-31)

- Fixed transpose bug in the usage of `acor` in `EnsembleSampler`.

1.0.0 (2012-02-15)

- Initial release.

Python Module Index

Index