emcee

Release 3.0.1

Oct 28, 2019

User Guide

1 Basic Usage **3**

2 How to Use This Guide **5**
- 2.1 Installation . . . 5
- 2.2 The Ensemble Sampler . . . 6
- 2.3 Moves . . . 9
- 2.4 Blobs . . . 11
- 2.5 Backends . . . 13
- 2.6 Autocorrelation Analysis . . . 17
- 2.7 Upgrading From Pre-3.0 Versions . . . 18
- 2.8 FAQ . . . 18
- 2.9 Quickstart . . . 19
- 2.10 Fitting a Model to Data . . . 21
- 2.11 Parallelization . . . 30
- 2.12 Autocorrelation analysis & convergence . . . 34
- 2.13 Saving & monitoring progress . . . 44

3 License & Attribution **51**

4 Changelog **53**
- 4.1 3.0.1 (2010-10-28) . . . 53
- 4.2 3.0.0 (2019-09-30) . . . 53
- 4.3 2.2.0 (2016-07-12) . . . 53
- 4.4 2.1.0 (2014-05-22) . . . 54
- 4.5 2.0.0 (2013-11-17) . . . 54
- 4.6 1.2.0 (2013-01-30) . . . 54
- 4.7 1.1.3 (2012-11-22) . . . 54
- 4.8 1.1.2 (2012-08-06) . . . 54
- 4.9 1.1.1 (2012-07-30) . . . 54
- 4.10 1.1.0 (2012-07-28) . . . 55
- 4.11 1.0.1 (2012-03-31) . . . 55
- 4.12 1.0.0 (2012-02-15) . . . 55

Python Module Index **57**

Index **59**

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 1

Basic Usage

If you wanted to draw samples from a 5 dimensional Gaussian, you would do something like:

```
import numpy as np
import emcee

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

ndim, nwalkers = 5, 100
ivar = 1. / np.random.rand(ndim)
p0 = np.random.randn(nwalkers, ndim)

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob, args=[ivar])
sampler.run_mcmc(p0, 10000)
```

A more complete example is available in the *Quickstart* tutorial.

CHAPTER 2

How to Use This Guide

To start, you're probably going to need to follow the *Installation* guide to get emcee installed on your computer. After you finish that, you can probably learn most of what you need from the tutorials listed below (you might want to start with *Quickstart* and go from there). If you need more details about specific functionality, the User Guide below should have what you need. If you run into any issues, please don't hesitate to open an issue on GitHub.

2.1 Installation

Since emcee is a pure Python module, it should be pretty easy to install. All you'll need numpy.

Note: For pre-release versions of emcee, you need to follow the instructions in *From source*.

2.1.1 Package managers

The easiest way to install the stable version of emcee is using pip or conda

```
pip install emcee
# or...
conda install -c conda-forge emcee
```

2.1.2 From source

emcee is developed on GitHub so if you feel like hacking or if you like all the most recent shininess, you can clone the source repository and install from there

```
git clone https://github.com/dfm/emcee.git
cd emcee
python setup.py install
```

2.1.3 Test the installation

To make sure that the installation went alright, you can execute some unit and integration tests. To do this, you'll need the source (see *From source* above) and py.test. You'll execute the tests by running the following command in the root directory of the source code:

```
py.test -v tests
```

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

2.2 The Ensemble Sampler

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

class emcee.**EnsembleSampler**(*nwalkers*, *ndim*, *log_prob_fn*, *pool=None*, *moves=None*, *args=None*, *kwargs=None*, *backend=None*, *vectorize=False*, *blobs_dtype=None*, *a=None*, *postargs=None*, *threads=None*, *live_dangerously=None*, *runtime_sortingfn=None*)

An ensemble MCMC sampler

Parameters

- **nwalkers** (*int*) – The number of walkers in the ensemble.
- **ndim** (*int*) – Number of dimensions in the parameter space.
- **log_prob_fn** (*callable*) – A function that takes a vector in the parameter space as input and returns the natural logarithm of the posterior probability (up to an additive constant) for that position.
- **moves** (*Optional*) – This can be a single move object, a list of moves, or a "weighted" list of the form `[(emcee.moves.StretchMove(), 0.1), ...]`. When running, the sampler will randomly select a move from this list (optionally with weights) for each proposal. (default: `StretchMove`)
- **args** (*Optional*) – A list of extra positional arguments for `log_prob_fn`. `log_prob_fn` will be called with the sequence `log_pprob_fn(p, *args, **kwargs)`.
- **kwargs** (*Optional*) – A dict of extra keyword arguments for `log_prob_fn`. `log_prob_fn` will be called with the sequence `log_pprob_fn(p, *args, **kwargs)`.
- **pool** (*Optional*) – An object with a `map` method that follows the same calling sequence as the built-in `map` function. This is generally used to compute the log-probabilities for the ensemble in parallel.
- **backend** (*Optional*) – Either a *`backends.Backend`* or a subclass (like *`backends.HDFBackend`*) that is used to store and serialize the state of the chain. By default, the chain is stored as a set of numpy arrays in memory, but new backends can be written to support other mediums.
- **vectorize** (*Optional[bool]*) – If `True`, `log_prob_fn` is expected to accept a list of position vectors instead of just one. Note that `pool` will be ignored if this is `True`. (default: `False`)

acceptance_fraction

The fraction of proposed steps that were accepted

compute_log_prob(*coords*)
Calculate the vector of log-probability for the walkers

Parameters **coords** – (ndarray[..., ndim]) The position vector in parameter space where the probability should be calculated.

This method returns:

- log_prob: A vector of log-probabilities with one entry for each walker in this sub-ensemble.
- blob: The list of meta data returned by the `log_post_fn` at this position or `None` if nothing was returned.

get_autocorr_time(***kwargs*)
Compute an estimate of the autocorrelation time for each parameter

Parameters

- **thin** (*Optional[int]*) – Use only every `thin` steps from the chain. The returned estimate is multiplied by `thin` so the estimated time is in units of steps, not thinned steps. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Other arguments are passed directly to *emcee.autocorr.integrated_time()*.

Returns

The integrated autocorrelation time estimate for the chain for each parameter.

Return type array[ndim]

get_blobs(***kwargs*)
Get the chain of blobs for each sample in the chain

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The chain of blobs.

Return type array[.., nwalkers]

get_chain(***kwargs*)
Get the stored chain of MCMC samples

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The MCMC samples.

Return type array[.., nwalkers, ndim]

get_last_sample(***kwargs*)
Access the most recent sample in the chain

get_log_prob(***kwargs*)
Get the chain of log probabilities evaluated at the MCMC samples

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The chain of log probabilities.

Return type array[.., nwalkers]

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()
Reset the bookkeeping parameters

run_mcmc(*initial_state*, *nsteps*, ***kwargs*)
Iterate *sample()* for `nsteps` iterations and return the result

Parameters

- **initial_state** – The initial state or position vector. Can also be `None` to resume from where :func:`run_mcmc` left off the last time it executed.
- **nsteps** – The number of steps to run.

Other parameters are directly passed to *sample()*.

This method returns the most recent result from *sample()*.

sample(*initial_state*, *log_prob0=None*, *rstate0=None*, *blobs0=None*, *iterations=1*, *tune=False*, *skip_initial_state_check=False*, *thin_by=1*, *thin=None*, *store=True*, *progress=False*)
Advance the chain as a generator

Parameters

- **initial_state** (*State or ndarray[nwalkers, ndim]*) – The initial *State* or positions of the walkers in the parameter space.
- **iterations** (*Optional[int]*) – The number of steps to generate.
- **tune** (*Optional[bool]*) – If `True`, the parameters of some moves will be automatically tuned.
- **thin_by** (*Optional[int]*) – If you only want to store and yield every `thin_by` samples in the chain, set `thin_by` to an integer greater than 1. When this is set, `iterations * thin_by` proposals will be made.
- **store** (*Optional[bool]*) – 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 analyze the samples after the fact (for burn-in for example) set `store` to `False`.
- **progress** (*Optional[bool or str]*) – If `True`, a progress bar will be shown as the sampler progresses. If a string, will select a specific `tqdm` progress bar - most notable is `'notebook'`, which shows a progress bar suitable for Jupyter notebooks. If `False`, no progress bar will be shown.

- **skip_initial_state_check** (*Optional[bool]*) – If `True`, a check that the initial_state can fully explore the space will be skipped. (default: `False`)

Every `thin_by` steps, this generator yields the *State* of the ensemble.

Note that several of the *EnsembleSampler* methods return or consume *State* objects:

class emcee.**State**(*coords*, *log_prob=None*, *blobs=None*, *random_state=None*, *copy=False*)

The state of the ensemble during an MCMC run

For backwards compatibility, this will unpack into `coords, log_prob, (blobs), random_state` when iterated over (where `blobs` will only be included if it exists and is not `None`).

Parameters

- **coords** (*ndarray[nwalkers, ndim]*) – The current positions of the walkers in the parameter space.
- **log_prob** (*ndarray[nwalkers, ndim], Optional*) – Log posterior probabilities for the walkers at positions given by `coords`.
- **blobs** (*Optional*) – The metadata "blobs" associated with the current position. The value is only returned if lnpostfn returns blobs too.
- **random_state** (*Optional*) – The current state of the random number generator.

2.3 Moves

emcee was originally built on the "stretch move" ensemble method from Goodman & Weare (2010), but starting with version 3, emcee nows allows proposals generated from a mixture of "moves". This can be used to get a more efficient sampler for models where the stretch move is not well suited, such as high dimensional or multi-modal probability surfaces.

A "move" is an algorithm for updating the coordinates of walkers in an ensemble sampler based on the current set of coordinates in a manner that satisfies detailed balance. In most cases, the update for each walker is based on the coordinates in some other set of walkers, the complementary ensemble.

These moves have been designed to update the ensemble in parallel following the prescription from Foreman-Mackey et al. (2013). This means that computationally expensive models can take advantage of multiple CPUs to accelerate sampling (see the *Parallelization* tutorial for more information).

The moves are selected using the `moves` keyword for the *EnsembleSampler* and the mixture can optionally be a weighted mixture of moves. During sampling, at each step, a move is randomly selected from the mixture and used as the proposal.

The default move is still the *moves.StretchMove*, but the others are described below. Many standard ensemble moves are available with parallelization provided by the *moves.RedBlueMove* abstract base class that implements the parallelization method described by Foreman-Mackey et al. (2013). In addition to these moves, there is also a framework for building Metropolis–Hastings proposals that update the walkers using independent proposals. *moves.MHMove* is the base class for this type of move and a concrete implementation of a Gaussian Metropolis proposal is found in *moves.GaussianMove*.

2.3.1 Ensemble moves

class emcee.moves.**RedBlueMove**(*nsplits=2*, *randomize_split=True*, *live_dangerously=False*)

An abstract red-blue ensemble move with parallelization as described in Foreman-Mackey et al. (2013).

Parameters

- **nsplits** (*Optional[int]*) – The number of sub-ensembles to use. Each sub-ensemble is updated in parallel using the other sets as the complementary ensemble. The default value is `2` and you probably won't need to change that.
- **randomize_split** (*Optional[bool]*) – Randomly shuffle walkers between sub-ensembles. The same number of walkers will be assigned to each sub-ensemble on each iteration. By default, this is `True`.
- **live_dangerously** (*Optional[bool]*) – By default, an update will fail with a `RuntimeError` if the number of walkers is smaller than twice the dimension of the problem because the walkers would then be stuck on a low dimensional subspace. This can be avoided by switching between the stretch move and, for example, a Metropolis-Hastings step. If you want to do this and suppress the error, set `live_dangerously = True`. Thanks goes (once again) to @dstndstn for this wonderful terminology.

propose(*model*, *state*)

Use the move to generate a proposal and compute the acceptance

Parameters

- **coords** – The initial coordinates of the walkers.
- **log_probs** – The initial log probabilities of the walkers.
- **log_prob_fn** – A function that computes the log probabilities for a subset of walkers.
- **random** – A numpy-compatible random number state.

class emcee.moves.**StretchMove**(*a=2.0*, ***kwargs*)

A Goodman & Weare (2010) "stretch move" with parallelization as described in Foreman-Mackey et al. (2013).

Parameters **a** – (optional) The stretch scale parameter. (default: `2.0`)

class emcee.moves.**WalkMove**(*s=None*, ***kwargs*)

A Goodman & Weare (2010) "walk move" with parallelization as described in Foreman-Mackey et al. (2013).

Parameters **s** – (optional) The number of helper walkers to use. By default it will use all the walkers in the complement.

class emcee.moves.**KDEMove**(*bw_method=None*, ***kwargs*)

A proposal using a KDE of the complementary ensemble

This is a simplified version of the method used in kombine. If you use this proposal, you should use *a lot* of walkers in your ensemble.

Parameters **bw_method** – The bandwidth estimation method. See the scipy docs for allowed values.

class emcee.moves.**DEMove**(*sigma=1e-05*, *gamma0=None*, ***kwargs*)

A proposal using differential evolution.

This Differential evolution proposal is implemented following Nelson et al. (2013).

Parameters

- **sigma** (*float*) – The standard deviation of the Gaussian used to stretch the proposal vector.
- **gamma0** (*Optional[float]*) – The mean stretch factor for the proposal vector. By default, it is $2.38/\sqrt{2\,\mathrm{ndim}}$ as recommended by the two references.

class emcee.moves.**DESnookerMove**(*gammas=1.7*, ***kwargs*)

A snooker proposal using differential evolution.

Based on Ter Braak & Vrugt (2008).

Credit goes to GitHub user mdanthony17 for proposing this as an addition to the original emcee package.

Parameters **gammas** (*Optional[float]*) – The mean stretch factor for the proposal vector. By default, it is 1.7 as recommended by the reference.

2.3.2 Metropolis–Hastings moves

class emcee.moves.**MHMove**(*proposal_function*, *ndim=None*)
A general Metropolis-Hastings proposal

Concrete implementations can be made by providing a `proposal_function` argument that implements the proposal as described below. For standard Gaussian Metropolis moves, `moves.GaussianMove` can be used.

Parameters

- **proposal_function** – The proposal function. It should take 2 arguments: a numpy-compatible random number generator and a `(K, ndim)` list of coordinate vectors. This function should return the proposed position and the log-ratio of the proposal probabilities ($\ln q(x;\, x') - \ln q(x';\, x)$ where x' is the proposed coordinate).
- **ndim** (*Optional[int]*) – If this proposal is only valid for a specific dimension of parameter space, set that here.

propose(*model*, *state*)
Use the move to generate a proposal and compute the acceptance

Parameters

- **coords** – The initial coordinates of the walkers.
- **log_probs** – The initial log probabilities of the walkers.
- **log_prob_fn** – A function that computes the log probabilities for a subset of walkers.
- **random** – A numpy-compatible random number state.

class emcee.moves.**GaussianMove**(*cov*, *mode='vector'*, *factor=None*)
A Metropolis step with a Gaussian proposal function.

Parameters

- **cov** – The covariance of the proposal function. This can be a scalar, vector, or matrix and the proposal will be assumed isotropic, axis-aligned, or general respectively.
- **mode** (*Optional*) – Select the method used for updating parameters. This can be one of `"vector"`, `"random"`, or `"sequential"`. The `"vector"` mode updates all dimensions simultaneously, `"random"` randomly selects a dimension and only updates that one, and `"sequential"` loops over dimensions and updates each one in turn.
- **factor** (*Optional[float]*) – If provided the proposal will be made with a standard deviation uniformly selected from the range `exp(U(-log(factor), log(factor))) * cov`. This is invalid for the `"vector"` mode.

Raises `ValueError` – If the proposal dimensions are invalid or if any of any of the other arguments are inconsistent.

2.4 Blobs

Way back in version 1.1 of emcee, the concept of blobs was introduced. This allows a user to track arbitrary metadata associated with every sample in the chain. The interface to access these blobs was previously a little clunky because it

was stored as a list of lists of blobs. In version 3, this interface has been updated to use NumPy arrays instead and the sampler will do type inference to save the simplest possible representation of the blobs.

2.4.1 Using blobs to track the value of the prior

A common pattern is to save the value of the log prior at every step in the chain. To do this, you could do something like:

```
import emcee
import numpy as np

def log_prior(params):
    return -0.5 * np.sum(params**2)

def log_like(params):
    return -0.5 * np.sum((params / 0.1)**2)

def log_prob(params):
    lp = log_prior(params)
    if not np.isfinite(lp):
        return -np.inf, -np.inf
    ll = log_like(params)
    if not np.isfinite(ll):
        return lp, -np.inf
    return lp + ll, lp

coords = np.random.randn(32, 3)
nwalkers, ndim = coords.shape
sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob)
sampler.run_mcmc(coords, 100)

log_prior_samps = sampler.get_blobs()
flat_log_prior_samps = sampler.get_blobs(flat=True)

print(log_prior_samps.shape)  # (100, 32)
print(flat_log_prior_samps.shape)  # (3200,)
```

After running this, the "blobs" stored by the sampler will be a `(nsteps, nwalkers)` NumPy array with the value of the log prior at every sample.

2.4.2 Named blobs & custom dtypes

If you want to save multiple pieces of metadata, it can be useful to name them. To implement this, we use the `blobs_dtype` argument in *`EnsembleSampler`*. For example, let's say that, for some reason, we wanted to save the mean of the parameters as well as the log prior. To do this, we would update the above example as follows:

```
def log_prob(params):
    lp = log_prior(params)
    if not np.isfinite(lp):
        return -np.inf, -np.inf
    ll = log_like(params)
    if not np.isfinite(ll):
        return lp, -np.inf
    return lp + ll, lp, np.mean(params)
```

(continues on next page)

(continued from previous page)

```
coords = np.random.randn(32, 3)
nwalkers, ndim = coords.shape

# Here are the important lines
dtype = [("log_prior", float), ("mean", float)]
sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob,
                                blobs_dtype=dtype)

sampler.run_mcmc(coords, 100)

blobs = sampler.get_blobs()
log_prior_samps = blobs["log_prior"]
mean_samps = blobs["mean"]
print(log_prior_samps.shape)
print(mean_samps.shape)

flat_blobs = sampler.get_blobs(flat=True)
flat_log_prior_samps = flat_blobs["log_prior"]
flat_mean_samps = flat_blobs["mean"]
print(flat_log_prior_samps.shape)
print(flat_mean_samps.shape)
```

This will print

```
(100, 32)
(100, 32)
(3200,)
(3200,)
```

and the `blobs` object will be a structured NumPy array with two columns called `log_prior` and `mean`.

2.5 Backends

Starting with version 3, emcee has an interface for serializing the sampler output. This can be useful in any scenario where you want to share the results of sampling or when sampling with an expensive model because, even if the sampler crashes, the current state of the chain will always be saved.

There is currently one backend that can be used to serialize the chain to a file: *emcee.backends.HDFBackend*. The methods and options for this backend are documented below. It can also be used as a reader for existing samplings. For example, if a chain was saved using the *backends.HDFBackend*, the results can be accessed as follows:

```
reader = emcee.backends.HDFBackend("chain_filename.h5", read_only=True)
flatchain = reader.get_chain(flat=True)
```

The `read_only` argument is not required, but it will make sure that you don't inadvertently overwrite the samples in the file.

class emcee.backends.**Backend**(*dtype=None*)

A simple default backend that stores the chain in memory

get_autocorr_time(*discard=0*, *thin=1*, ***kwargs*)

Compute an estimate of the autocorrelation time for each parameter

Parameters

- **thin** (*Optional[int]*) – Use only every `thin` steps from the chain. The returned estimate is multiplied by `thin` so the estimated time is in units of steps, not thinned steps. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Other arguments are passed directly to *emcee.autocorr.integrated_time()*.

Returns

The integrated autocorrelation time estimate for the chain for each parameter.

Return type array[ndim]

get_blobs(***kwargs*)
Get the chain of blobs for each sample in the chain

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The chain of blobs.

Return type array[.., nwalkers]

get_chain(***kwargs*)
Get the stored chain of MCMC samples

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The MCMC samples.

Return type array[.., nwalkers, ndim]

get_last_sample()
Access the most recent sample in the chain

get_log_prob(***kwargs*)
Get the chain of log probabilities evaluated at the MCMC samples

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The chain of log probabilities.

Return type array[.., nwalkers]

grow(*ngrow*, *blobs*)
Expand the storage space by some number of samples

Parameters

- **ngrow** (*int*) – The number of steps to grow the chain.
- **blobs** – The current list of blobs. This is used to compute the dtype for the blobs array.

has_blobs()
Returns `True` if the model includes blobs

reset(*nwalkers*, *ndim*)
Clear the state of the chain and empty the backend

Parameters

- **nwakers** (*int*) – The size of the ensemble
- **ndim** (*int*) – The number of dimensions

save_step(*state*, *accepted*)
Save a step to the backend

Parameters

- **state** (`State`) – The `State` of the ensemble.
- **accepted** (*ndarray*) – An array of boolean flags indicating whether or not the proposal for each walker was accepted.

shape
The dimensions of the ensemble `(nwalkers, ndim)`

class emcee.backends.**HDFBackend**(*filename*, *name='mcmc'*, *read_only=False*, *dtype=None*)
A backend that stores the chain in an HDF5 file using h5py

Note: You must install h5py to use this backend.

Parameters

- **filename** (*str*) – The name of the HDF5 file where the chain will be saved.
- **name** (*str; optional*) – The name of the group where the chain will be saved.
- **read_only** (*bool; optional*) – If `True`, the backend will throw a `RuntimeError` if the file is opened with write access.

get_autocorr_time(*discard=0*, *thin=1*, ***kwargs*)
Compute an estimate of the autocorrelation time for each parameter

Parameters

- **thin** (*Optional[int]*) – Use only every `thin` steps from the chain. The returned estimate is multiplied by `thin` so the estimated time is in units of steps, not thinned steps. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Other arguments are passed directly to *emcee.autocorr.integrated_time()*.

Returns

The integrated autocorrelation time estimate for the chain for each parameter.

Return type array[ndim]

get_blobs(***kwargs*)
Get the chain of blobs for each sample in the chain

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The chain of blobs.

Return type array[.., nwalkers]

get_chain(***kwargs*)
Get the stored chain of MCMC samples

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The MCMC samples.

Return type array[.., nwalkers, ndim]

get_last_sample()
Access the most recent sample in the chain

get_log_prob(***kwargs*)
Get the chain of log probabilities evaluated at the MCMC samples

Parameters

- **flat** (*Optional[bool]*) – Flatten the chain across the ensemble. (default: `False`)
- **thin** (*Optional[int]*) – Take only every `thin` steps from the chain. (default: `1`)
- **discard** (*Optional[int]*) – Discard the first `discard` steps in the chain as burn-in. (default: `0`)

Returns The chain of log probabilities.

Return type array[.., nwalkers]

grow(*ngrow*, *blobs*)
Expand the storage space by some number of samples

Parameters

- **ngrow** (*int*) – The number of steps to grow the chain.
- **blobs** – The current list of blobs. This is used to compute the dtype for the blobs array.

has_blobs()
Returns `True` if the model includes blobs

reset(*nwalkers*, *ndim*)
Clear the state of the chain and empty the backend

Parameters

- **nwakers** (*int*) – The size of the ensemble
- **ndim** (*int*) – The number of dimensions

save_step(*state*, *accepted*)
Save a step to the backend

Parameters

- **state** (State) – The State of the ensemble.
- **accepted** (*ndarray*) – An array of boolean flags indicating whether or not the proposal for each walker was accepted.

shape
The dimensions of the ensemble (nwalkers, ndim)

2.6 Autocorrelation Analysis

A good heuristic for assessing convergence of samplings is the integrated autocorrelation time. emcee includes tools for computing this and the autocorrelation function itself. More details can be found in *Autocorrelation analysis & convergence*.

emcee.autocorr.**integrated_time**(*x*, *c=5*, *tol=50*, *quiet=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.

Parameters

- **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.
- **c** (*Optional[float]*) – The step size for the window search. (default: 5)
- **tol** (*Optional[float]*) – The minimum number of autocorrelation times needed to trust the estimate. (default: 50)
- **quiet** (*Optional[bool]*) – This argument controls the behavior when the chain is too short. If True, give a warning instead of raising an AutocorrError. (default: False)

Returns

An estimate of the integrated autocorrelation time of the time series x computed along the axis axis.

Return type float or array

Raises

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

emcee.autocorr.**function_1d**(*x*)
Estimate the normalized autocorrelation function of a 1-D series

Parameters **x** – The series as a 1-D numpy array.

Returns The autocorrelation function of the time series.

Return type array

2.7 Upgrading From Pre-3.0 Versions

The version 3 release of emcee is the biggest update in years. That being said, we've made every attempt to maintain backwards compatibility while still offering new features. The main new features include:

1. A *Moves* interface that allows the use of a variety of ensemble proposals,
2. A more self consistent and user-friendly *Blobs* interface,
3. A *Backends* interface that simplifies the process of serializing the sampling results, and
4. The long requested progress bar (implemented using tqdm) so that users can watch the grass grow while the sampler does its thing (this is as simple as installing tqdm and setting `progress=True` in *`EnsembleSampler`*).

To improve the stability and supportability of emcee, we also removed some features. The main removals are as follows:

1. The `threads` keyword argument has been removed in favor of the `pool` interface (see the *Parallelization* tutorial for more information). The old interface had issues with memory consumption and hanging processes when the sampler object wasn't explicitly deleted. The `pool` interface has been supported since the first release of emcee and existing code should be easy to update following the *Parallelization* tutorial.
2. The `MPIPool` has been removed and forked to the schwimmbad project. There was a longstanding issue with memory leaks and random crashes of the emcee implementation of the `MPIPool` that have been fixed in schwimmbad. schwimmbad also supports several other `pool` interfaces that can be used for parallel sampling. See the *Parallelization* tutorial for more details.
3. The `PTSampler` has been removed and forked to the ptemcee project. The existing implementation had been gathering dust and there aren't enough resources to maintain the sampler within the emcee project.

2.8 FAQ

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

2.8.1 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.

2.8.2 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.

2.8.3 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: This tutorial was generated from an IPython notebook that can be downloaded here.

2.9 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.

2.9.1 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
```

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 `log_prob`:

```
def log_prob(x, mu, cov):
    diff = x - mu
    return -0.5 * np.dot(diff, np.linalg.solve(cov, diff))
```

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 5 dimensions:

```
ndim = 5

np.random.seed(42)
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.

How about we use 32 walkers? Before we go on, we need to guess a starting point for each of the 32 walkers. This position will be a 5-dimensional vector so the initial guess should be a 32-by-5 array. It's not a very good guess but we'll just guess a random number between 0 and 1 for each component:

```
nwalkers = 32
p0 = np.random.rand(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:

```
import emcee

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob, args=[means, cov])
```

Remember how our function `log_prob` 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:

```
log_prob(p0[0], means, cov)
```

```
-2.5960945890854434
```

If we didn't provide any `args` parameter, the calling sequence would be `log_prob(p0[0])` 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`:

```
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 `state`. 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 bookkeeping 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.

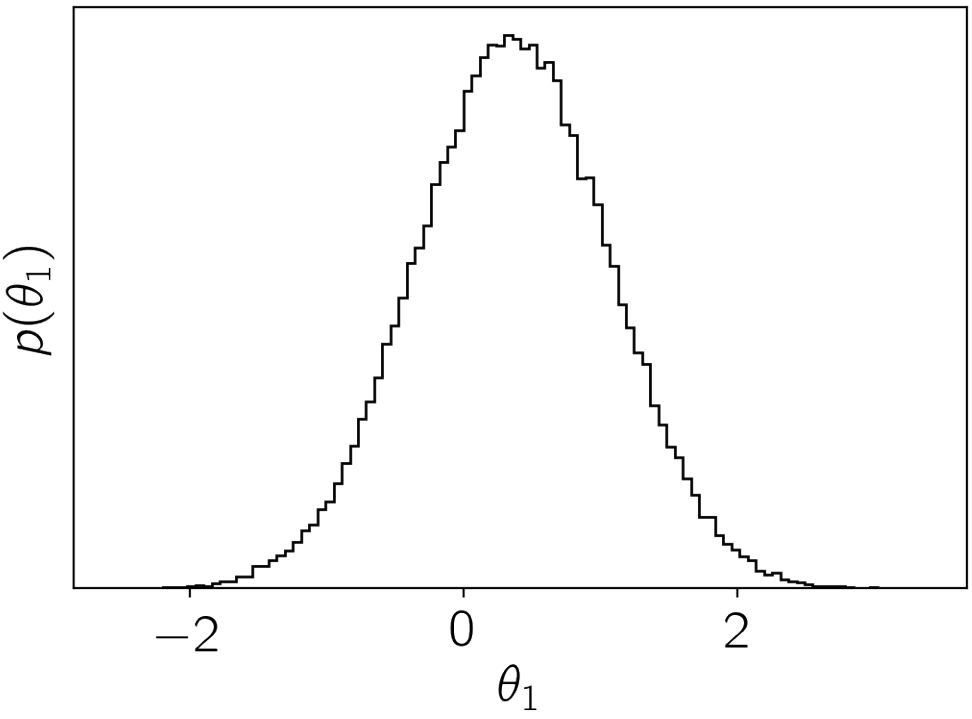
Now, we can do our production run of 10000 steps:

```
sampler.run_mcmc(state, 10000);
```

The samples can be accessed using the *`EnsembleSampler.get_chain()`* method. This will return an array with the shape `(10000, 32, 5)` giving the parameter values for each walker at each step in the chain. Take note of that shape and make sure that you know where each of those numbers come from. You can make histograms of these samples to get an estimate of the density that you were sampling:

```
import matplotlib.pyplot as plt

samples = sampler.get_chain(flat=True)
plt.hist(samples[:, 0], 100, color="k", histtype="step")
plt.xlabel(r"$\theta_1$")
plt.ylabel(r"$p(\theta_1)$")
plt.gca().set_yticks([]);
```



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)))
```

```
Mean acceptance fraction: 0.552
```

and the integrated autocorrelation time (see the *Autocorrelation analysis & convergence* tutorial for more details)

```
print(
    "Mean autocorrelation time: {0:.3f} steps".format(
        np.mean(sampler.get_autocorr_time())
    )
)
```

```
Mean autocorrelation time: 57.112 steps
```

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

2.10 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.

2.10.1 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:

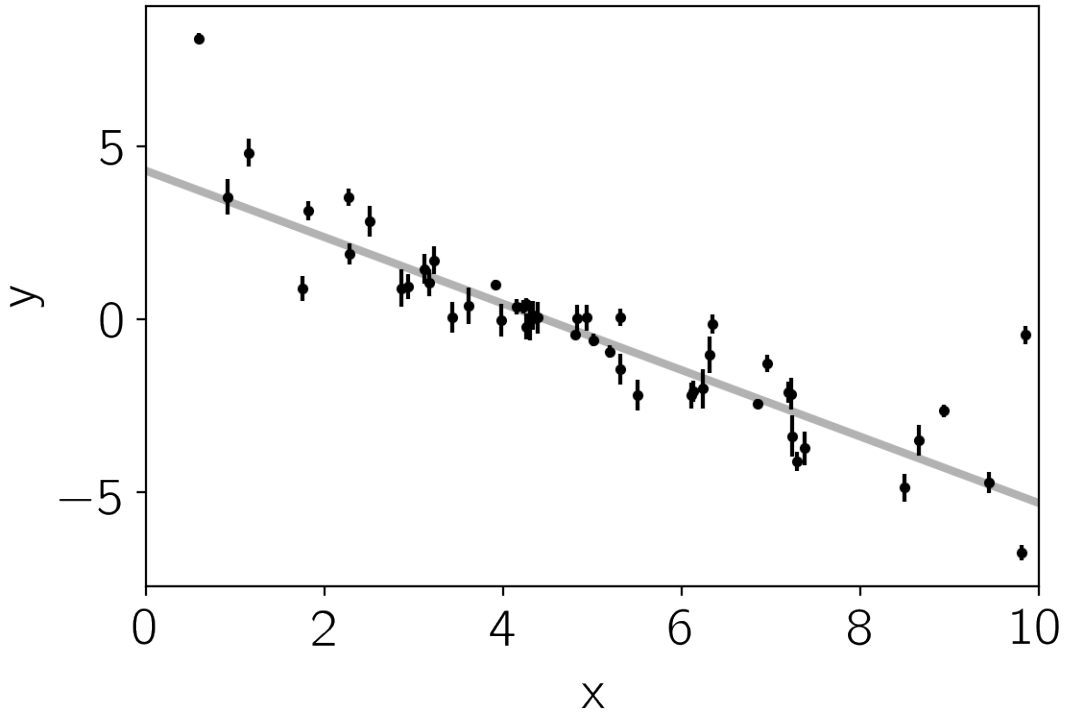
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(123)

# 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)

plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
x0 = np.linspace(0, 10, 500)
plt.plot(x0, m_true * x0 + b_true, "k", alpha=0.3, lw=3)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");
```

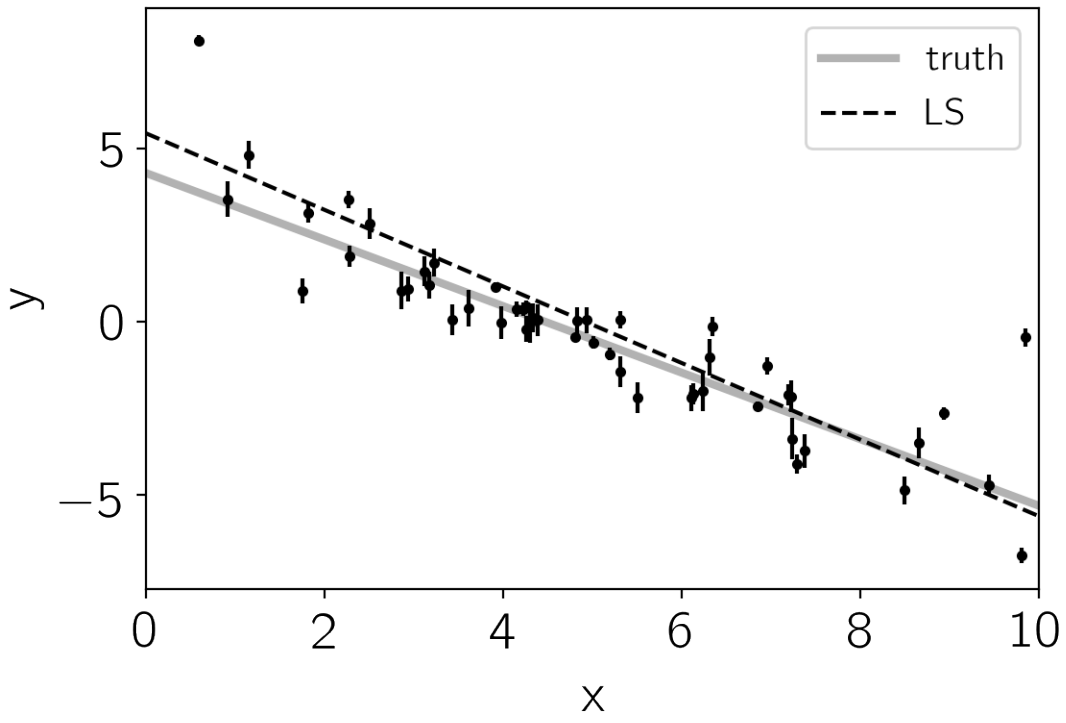


The true model is shown as the thick grey line and the effect of the underestimated uncertainties is obvious when you look at this figure. The standard way to fit a line to these data (assuming independent Gaussian error bars) is linear least squares. Linear least squares is appealing because solving 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.vander(x, 2)
C = np.diag(yerr * yerr)
ATA = np.dot(A.T, A / (yerr ** 2)[:, None])
cov = np.linalg.inv(ATA)
w = np.linalg.solve(ATA, np.dot(A.T, y / yerr ** 2))
print("Least-squares estimates:")
print("m = {0:.3f} ± {1:.3f}".format(w[0], np.sqrt(cov[0, 0])))
print("b = {0:.3f} ± {1:.3f}".format(w[1], np.sqrt(cov[1, 1])))

plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
plt.plot(x0, m_true * x0 + b_true, "k", alpha=0.3, lw=3, label="truth")
plt.plot(x0, np.dot(np.vander(x0, 2), w), "--k", label="LS")
plt.legend(fontsize=14)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");
```

```
Least-squares estimates:
m = -1.104 ± 0.016
b = 5.441 ± 0.091
```



This figure shows the least-squares estimate of the line parameters 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).

2.10.2 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 log_likelihood(theta, x, y, yerr):
    m, b, log_f = theta
    model = m * x + b
    sigma2 = yerr ** 2 + model ** 2 * np.exp(2 * log_f)
    return -0.5 * np.sum((y - model) ** 2 / sigma2 + np.log(sigma2))
```

In this code snippet, you'll notice that we're 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:

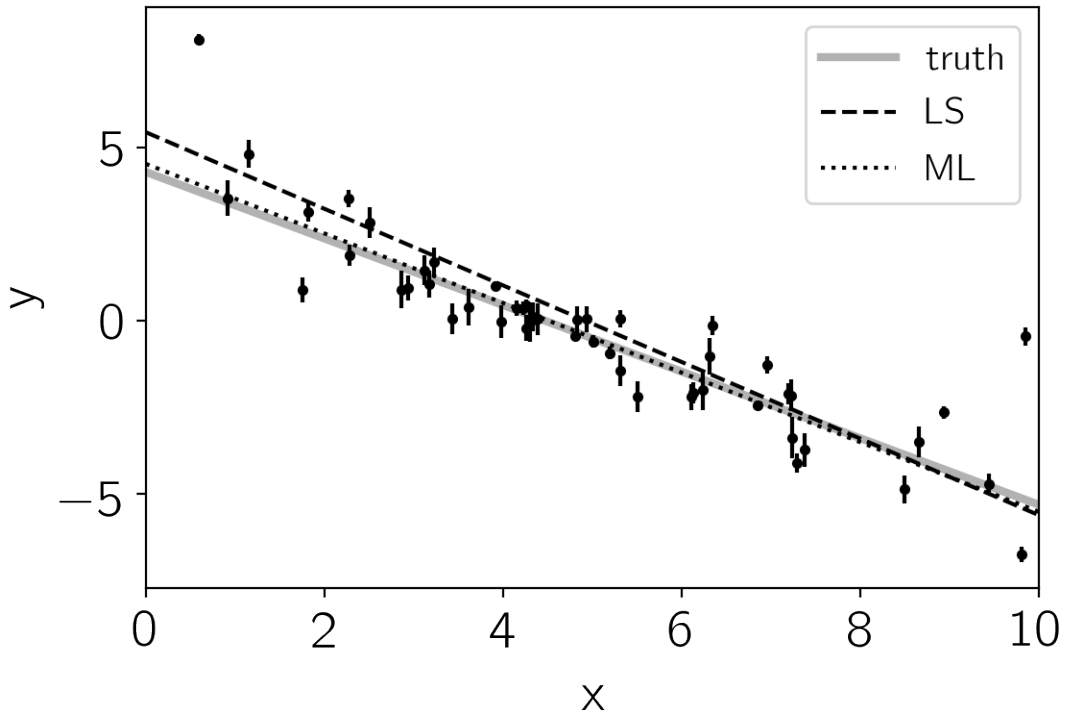
```
from scipy.optimize import minimize

np.random.seed(42)
nll = lambda *args: -log_likelihood(*args)
initial = np.array([m_true, b_true, np.log(f_true)]) + 0.1 * np.random.randn(3)
soln = minimize(nll, initial, args=(x, y, yerr))
m_ml, b_ml, log_f_ml = soln.x

print("Maximum likelihood estimates:")
print("m = {0:.3f}".format(m_ml))
print("b = {0:.3f}".format(b_ml))
print("f = {0:.3f}".format(np.exp(log_f_ml)))

plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
plt.plot(x0, m_true * x0 + b_true, "k", alpha=0.3, lw=3, label="truth")
plt.plot(x0, np.dot(np.vander(x0, 2), w), "--k", label="LS")
plt.plot(x0, np.dot(np.vander(x0, 2), [m_ml, b_ml]), ":k", label="ML")
plt.legend(fontsize=14)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");
```

```
Maximum likelihood estimates:
m = -1.003
b = 4.528
f = 0.454
```



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). In this figure, the maximum likelihood (ML) result is plotted as a dotted black line—compared to the true model (grey line) and linear least-squares (LS; dashed line). That looks better!

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.

2.10.3 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):

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

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

```
def log_probability(theta, x, y, yerr):
    lp = log_prior(theta)
    if not np.isfinite(lp):
        return -np.inf
    return lp + log_likelihood(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) and then run 5,000 steps of MCMC.

```
import emcee

pos = soln.x + 1e-4 * np.random.randn(32, 3)
nwalkers, ndim = pos.shape

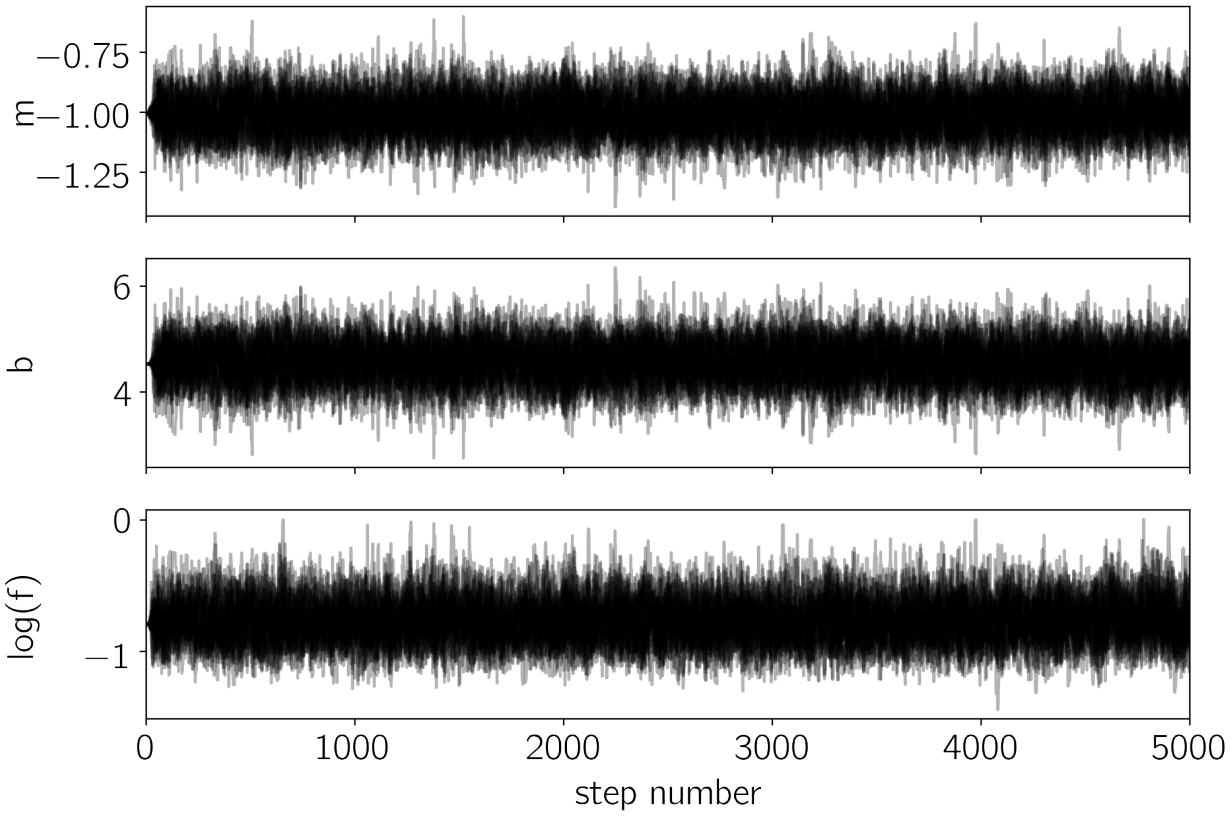
sampler = emcee.EnsembleSampler(nwalkers, ndim, log_probability, args=(x, y, yerr))
sampler.run_mcmc(pos, 5000, progress=True);
```

```
100%|| 5000/5000 [00:06<00:00, 815.49it/s]
```

Let's take a look at what the sampler has done. A good first step is to look at the time series of the parameters in the chain. The samples can be accessed using the *`EnsembleSampler.get_chain()`* method. This will return an array with the shape `(5000, 32, 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:

```
fig, axes = plt.subplots(3, figsize=(10, 7), sharex=True)
samples = sampler.get_chain()
labels = ["m", "b", "log(f)"]
for i in range(ndim):
    ax = axes[i]
    ax.plot(samples[:, :, i], "k", alpha=0.3)
    ax.set_xlim(0, len(samples))
    ax.set_ylabel(labels[i])
    ax.yaxis.set_label_coords(-0.1, 0.5)

axes[-1].set_xlabel("step number");
```



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 we can look at an estimate of the integrated autocorrelation time (see the *Autocorrelation analysis & convergence* tutorial for more details):

```
tau = sampler.get_autocorr_time()
print(tau)
```

```
[35.73919335 35.69339914 36.05722561]
```

This suggests that only about 40 steps are needed for the chain to "forget" where it started. It's not unreasonable to throw away a few times this number of steps as "burn-in". Let's discard the initial 100 steps, thin by about half the autocorrelation time (15 steps), and flatten the chain so that we have a flat list of samples:

```
flat_samples = sampler.get_chain(discard=100, thin=15, flat=True)
print(flat_samples.shape)
```

```
(10432, 3)
```

2.10.4 Results

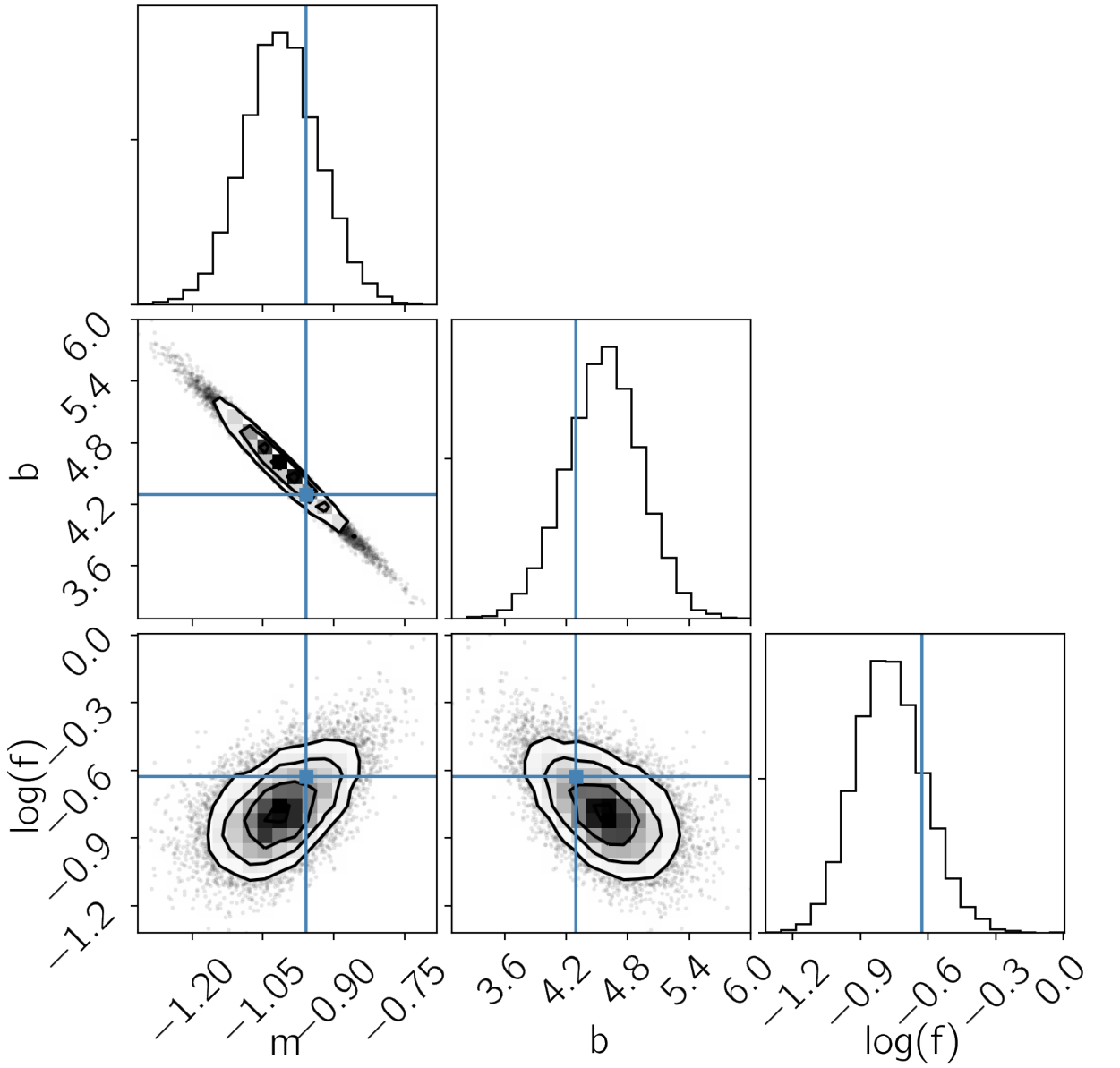
Now that we have this list of samples, let's make one of the most useful plots you can make with your MCMC results: *a corner plot*. You'll need the corner.py module but once you have it, generating a corner plot is as simple as:

```
import corner
```

(continues on next page)

(continued from previous page)

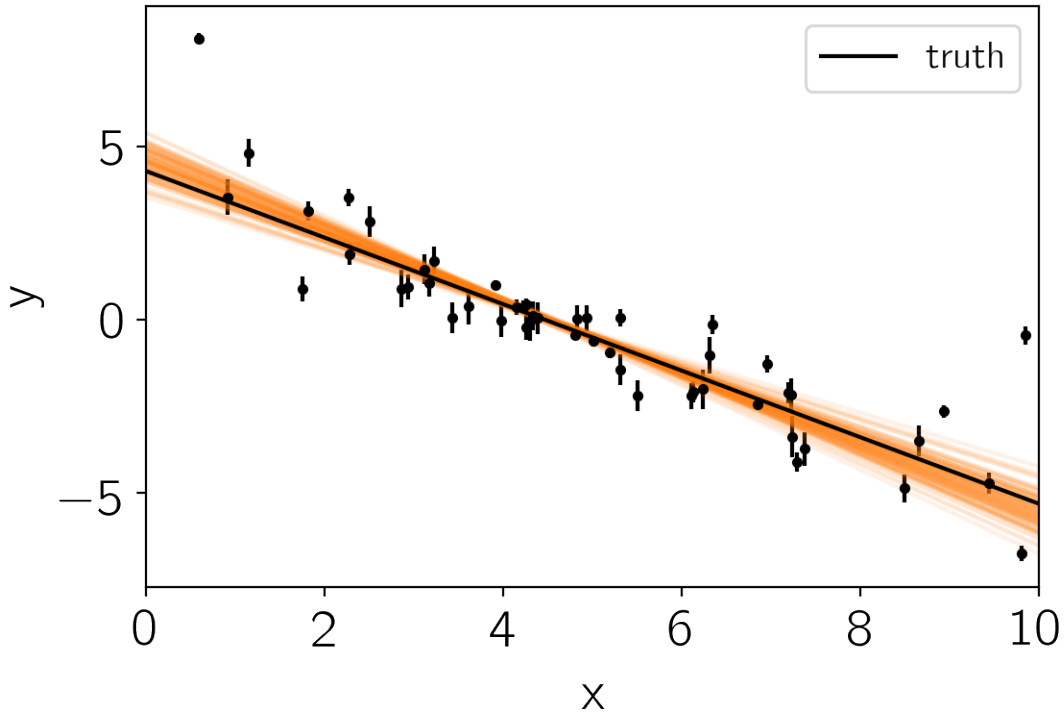
```
fig = corner.corner(
    flat_samples, labels=labels, truths=[m_true, b_true, np.log(f_true)]
);
```



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:

```
inds = np.random.randint(len(flat_samples), size=100)
for ind in inds:
    sample = flat_samples[ind]
    plt.plot(x0, np.dot(np.vander(x0, 2), sample[:2]), "C1", alpha=0.1)
plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
plt.plot(x0, m_true * x0 + b_true, "k", label="truth")
plt.legend(fontsize=14)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");
```



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:

```
from IPython.display import display, Math

for i in range(ndim):
    mcmc = np.percentile(flat_samples[:, i], [16, 50, 84])
    q = np.diff(mcmc)
    txt = "\mathrm{{{3}}} = {0:.3f}_{{-{1:.3f}}}^{{{2:.3f}}}"
    txt = txt.format(mcmc[1], q[0], q[1], labels[i])
    display(Math(txt))
```

$$\mathrm{m} = -1.012^{0.081}_{-0.078}$$

$$\mathrm{b} = 4.566^{0.355}_{-0.372}$$

$$\log(\mathrm{f}) = -0.776^{0.162}_{-0.147}$$

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

2.11 Parallelization

Note: Some builds of NumPy (including the version included with Anaconda) will automatically parallelize some operations using something like the MKL linear algebra. This can cause problems when used with the parallelization methods described here so it can be good to turn that off (by setting the environment variable `OMP_NUM_THREADS=1`, for example).

```
import os

os.environ["OMP_NUM_THREADS"] = "1"
```

With emcee, it's easy to make use of multiple CPUs to speed up slow sampling. There will always be some computational overhead introduced by parallelization so it will only be beneficial in the case where the model is expensive, but this is often true for real research problems. All parallelization techniques are accessed using the `pool` keyword argument in the *`EnsembleSampler`* class but, depending on your system and your model, there are a few pool options that you can choose from. In general, a `pool` is any Python object with a `map` method that can be used to apply a function to a list of numpy arrays. Below, we will discuss a few options.

In all of the following examples, we'll test the code with the following convoluted model:

```
import time
import numpy as np


def log_prob(theta):
    t = time.time() + np.random.uniform(0.005, 0.008)
    while True:
        if time.time() >= t:
            break
    return -0.5 * np.sum(theta ** 2)
```

This probability function will randomly sleep for a fraction of a second every time it is called. This is meant to emulate a more realistic situation where the model is computationally expensive to compute.

To start, let's sample the usual (serial) way:

```
import emcee

np.random.seed(42)
initial = np.random.randn(32, 5)
nwalkers, ndim = initial.shape
nsteps = 100

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob)
start = time.time()
sampler.run_mcmc(initial, nsteps, progress=True)
end = time.time()
serial_time = end - start
print("Serial took {0:.1f} seconds".format(serial_time))
```

```
100%|| 100/100 [00:21<00:00,  4.68it/s]
```

```
Serial took 21.4 seconds
```

2.11.1 Multiprocessing

The simplest method of parallelizing emcee is to use the multiprocessing module from the standard library. To parallelize the above sampling, you could update the code as follows:

```
from multiprocessing import Pool

with Pool() as pool:
    sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob, pool=pool)
    start = time.time()
    sampler.run_mcmc(initial, nsteps, progress=True)
    end = time.time()
    multi_time = end - start
    print("Multiprocessing took {0:.1f} seconds".format(multi_time))
    print("{0:.1f} times faster than serial".format(serial_time / multi_time))
```

```
100%|| 100/100 [00:06<00:00, 16.42it/s]
```

```
Multiprocessing took 6.4 seconds
3.4 times faster than serial
```

I have 4 cores on the machine where this is being tested:

```
from multiprocessing import cpu_count

ncpu = cpu_count()
print("{0} CPUs".format(ncpu))
```

```
4 CPUs
```

We don't quite get the factor of 4 runtime decrease that you might expect because there is some overhead in the parallelization, but we're getting pretty close with this example and this will get even closer for more expensive models.

2.11.2 MPI

Multiprocessing can only be used for distributing calculations across processors on one machine. If you want to take advantage of a bigger cluster, you'll need to use MPI. In that case, you need to execute the code using the `mpiexec` executable, so this demo is slightly more convoluted. For this example, we'll write the code to a file called `script.py` and then execute it using MPI, but when you really use the MPI pool, you'll probably just want to edit the script directly. To run this example, you'll first need to install the schwimmbad library because emcee no longer includes its own `MPIPool`.

```
with open("script.py", "w") as f:
    f.write("""
import sys
import time
import emcee
import numpy as np
from schwimmbad import MPIPool

def log_prob(theta):
    t = time.time() + np.random.uniform(0.005, 0.008)
    while True:
```

(continues on next page)

(continued from previous page)

```
        if time.time() >= t:
            break
    return -0.5*np.sum(theta**2)

with MPIPool() as pool:
    if not pool.is_master():
        pool.wait()
        sys.exit(0)

    np.random.seed(42)
    initial = np.random.randn(32, 5)
    nwalkers, ndim = initial.shape
    nsteps = 100

    sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob, pool=pool)
    start = time.time()
    sampler.run_mcmc(initial, nsteps)
    end = time.time()
    print(end - start)
""")

mpi_time = !mpiexec -n {ncpu} python script.py
mpi_time = float(mpi_time[0])
print("MPI took {0:.1f} seconds".format(mpi_time))
print("{0:.1f} times faster than serial".format(serial_time / mpi_time))
```

```
MPI took 8.8 seconds
2.4 times faster than serial
```

There is often more overhead introduced by MPI than multiprocessing so we get less of a gain this time. That being said, MPI is much more flexible and it can be used to scale to huge systems.

2.11.3 Pickling, data transfer & arguments

All parallel Python implementations work by spinning up multiple `python` processes with identical environments then and passing information between the processes using `pickle`. This means that the probability function must be picklable.

Some users might hit issues when they use `args` to pass data to their model. These args must be pickled and passed every time the model is called. This can be a problem if you have a large dataset, as you can see here:

```
def log_prob_data(theta, data):
    a = data[0]  # Use the data somehow...
    t = time.time() + np.random.uniform(0.005, 0.008)
    while True:
        if time.time() >= t:
            break
    return -0.5 * np.sum(theta ** 2)


data = np.random.randn(5000, 200)

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob_data, args=(data,))
start = time.time()
sampler.run_mcmc(initial, nsteps, progress=True)
```

(continues on next page)

(continued from previous page)

```
end = time.time()
serial_data_time = end - start
print("Serial took {0:.1f} seconds".format(serial_data_time))
```

```
100%|| 100/100 [00:21<00:00,  4.53it/s]
```

```
Serial took 21.5 seconds
```

We basically get no change in performance when we include the `data` argument here. Now let's try including this naively using multiprocessing:

```
with Pool() as pool:
    sampler = emcee.EnsembleSampler(
        nwalkers, ndim, log_prob_data, pool=pool, args=(data,)
    )
    start = time.time()
    sampler.run_mcmc(initial, nsteps, progress=True)
    end = time.time()
    multi_data_time = end - start
    print("Multiprocessing took {0:.1f} seconds".format(multi_data_time))
    print(
        "{0:.1f} times faster(?) than serial".format(serial_data_time / multi_data_
↪time)
    )
```

```
100%|| 100/100 [01:08<00:00,  1.63it/s]
```

```
Multiprocessing took 68.6 seconds
0.3 times faster(?) than serial
```

Brutal.

We can do better than that though. It's a bit ugly, but if we just make `data` a global variable and use that variable within the model calculation, then we take no hit at all.

```
def log_prob_data_global(theta):
    a = data[0]  # Use the data somehow...
    t = time.time() + np.random.uniform(0.005, 0.008)
    while True:
        if time.time() >= t:
            break
    return -0.5 * np.sum(theta ** 2)


with Pool() as pool:
    sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob_data_global, pool=pool)
    start = time.time()
    sampler.run_mcmc(initial, nsteps, progress=True)
    end = time.time()
    multi_data_global_time = end - start
    print("Multiprocessing took {0:.1f} seconds".format(multi_data_global_time))
    print(
        "{0:.1f} times faster than serial".format(
            serial_data_time / multi_data_global_time
        )
    )
```

```
100%|| 100/100 [00:06<00:00, 16.29it/s]
```

```
Multiprocessing took 6.4 seconds
3.4 times faster than serial
```

That's better! This works because, in the global variable case, the dataset is only pickled and passed between processes once (when the pool is created) instead of once for every model evaluation. **Note:** This tutorial was generated from an IPython notebook that can be downloaded here.

2.12 Autocorrelation analysis & convergence

In this tutorial, we will discuss a method for convincing yourself that your chains are sufficiently converged. This can be a difficult subject to discuss because it isn't formally possible to guarantee convergence for any but the simplest models, and therefore any argument that you make will be circular and heuristic. However, some discussion of autocorrelation analysis is (or should be!) a necessary part of any publication using MCMC.

With emcee, we follow Goodman & Weare (2010) and recommend using the *integrated autocorrelation time* to quantify the effects of sampling error on your results. The basic idea is that the samples in your chain are not independent and you must estimate the effective number of independent samples. There are other convergence diagnostics like the Gelman–Rubin statistic (*Note: you should not compute the G–R statistic using multiple chains in the same emcee ensemble because the chains are not independent!*) but, since the integrated autocorrelation time directly quantifies the Monte Carlo error (and hence the efficiency of the sampler) on any integrals computed using the MCMC results, it is the natural quantity of interest when judging the robustness of an MCMC analysis.

2.12.1 Monte Carlo error

The goal of every MCMC analysis is to evaluate integrals of the form

$$\mathrm{E}_{p(\theta)}[f(\theta)] = \int f(\theta)\, p(\theta)\, \mathrm{d}\theta \quad .$$

If you had some way of generating N samples $\theta^{(n)}$ from the probability density $p(\theta)$, then you could approximate this integral as

$$\mathrm{E}_{p(\theta)}[f(\theta)] \approx \frac{1}{N} \sum_{n=1}^{N} f(\theta^{(n)})$$

where the sum is over the samples from $p(\theta)$. If these samples are independent, then the sampling variance on this estimator is

$$\sigma^2 = \frac{1}{N} \mathrm{Var}_{p(\theta)}[f(\theta)]$$

and the error decreases as $1/\sqrt{N}$ as you generate more samples. In the case of MCMC, the samples are not independent and the error is actually given by

$$\sigma^2 = \frac{\tau_f}{N} \mathrm{Var}_{p(\theta)}[f(\theta)]$$

where τ_f is the *integrated autocorrelation time* for the chain $f(\theta^{(n)})$. In other words, N/τ_f is the effective number of samples and τ_f is the number of steps that are needed before the chain "forgets" where it started. This means that, if you can estimate τ_f, then you can estimate the number of samples that you need to generate to reduce the relative error on your target integral to (say) a few percent.

Note: It is important to remember that τ_f depends on the specific function $f(\theta)$. This means that there isn't just *one* integrated autocorrelation time for a given Markov chain. Instead, you must compute a different τ_f for any integral you estimate using the samples.

2.12.2 Computing autocorrelation times

There is a great discussion of methods for autocorrelation estimation in a set of lecture notes by Alan Sokal and the interested reader should take a look at that for a more formal discussion, but I'll include a summary of some of the relevant points here. The integrated autocorrelation time is defined as

$$\tau_f = \sum_{\tau=-\infty}^{\infty} \rho_f(\tau)$$

where $\rho_f(\tau)$ is the normalized autocorrelation function of the stochastic process that generated the chain for f. You can estimate $\rho_f(\tau)$ using a finite chain $\{f_n\}_{n=1}^N$ as

$$\hat{\rho}_f(\tau) = \hat{c}_f(\tau)/\hat{c}_f(0)$$

where

$$\hat{c}_f(\tau) = \frac{1}{N-\tau} \sum_{n=1}^{N-\tau} (f_n - \mu_f)\,(f_{n+\tau} - \mu_f)$$

and

$$\mu_f = \frac{1}{N} \sum_{n=1}^{N} f_n \quad .$$

(Note: In practice, it is actually more computationally efficient to compute $\hat{c}_f(\tau)$ using a fast Fourier transform than summing it directly.)

Now, you might expect that you can estimate τ_f using this estimator for $\rho_f(\tau)$ as

$$\hat{\tau}_f \stackrel{?}{=} \sum_{\tau=-N}^{N} \hat{\rho}_f(\tau) = 1 + 2\sum_{\tau=1}^{N} \hat{\rho}_f(\tau)$$

but this isn't actually a very good idea. At longer lags, $\hat{\rho}_f(\tau)$ starts to contain more noise than signal and summing all the way out to N will result in a very noisy estimate of τ_f. Instead, we want to estimate τ_f as

$$\hat{\tau}_f(M) = 1 + 2\sum_{\tau=1}^{M} \hat{\rho}_f(\tau)$$

for some $M \ll N$. As discussed by Sokal in the notes linked above, the introduction of M decreases the variance of the estimator at the cost of some added bias and he suggests choosing the smallest value of M where $M \geq C\,\hat{\tau}_f(M)$ for a constant $C \sim 5$. Sokal says that he finds this procedure to work well for chains longer than $1000\,\tau_f$, but the situation is a bit better with emcee because we can use the parallel chains to reduce the variance and we've found that chains longer than about $50\,\tau$ are often sufficient.

2.12.3 A toy problem

To demonstrate this method, we'll start by generating a set of "chains" from a process with known autocorrelation structure. To generate a large enough dataset, we'll use celerite:

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1234)
```

(continues on next page)

(continued from previous page)

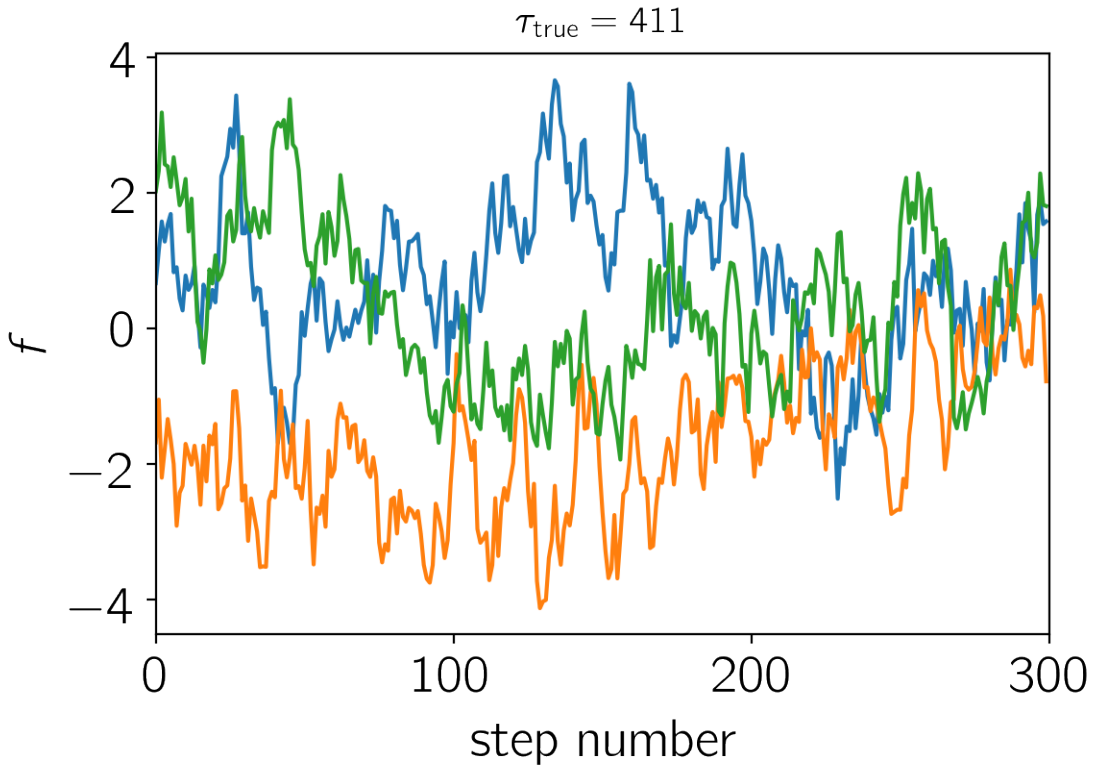
```
# Build the celerite model:
import celerite
from celerite import terms

kernel = terms.RealTerm(log_a=0.0, log_c=-6.0)
kernel += terms.RealTerm(log_a=0.0, log_c=-2.0)

# The true autocorrelation time can be calculated analytically:
true_tau = sum(2 * np.exp(t.log_a - t.log_c) for t in kernel.terms)
true_tau /= sum(np.exp(t.log_a) for t in kernel.terms)
true_tau

# Simulate a set of chains:
gp = celerite.GP(kernel)
t = np.arange(2000000)
gp.compute(t)
y = gp.sample(size=32)

# Let's plot a little segment with a few samples:
plt.plot(y[:3, :300].T)
plt.xlim(0, 300)
plt.xlabel("step number")
plt.ylabel("$f$")
plt.title("$\\tau_\mathrm{{true}} = {0:.0f}$".format(true_tau), fontsize=14);
```



Now we'll estimate the empirical autocorrelation function for each of these parallel chains and compare this to the true function.

```
def next_pow_two(n):
    i = 1
    while i < n:
        i = i << 1
    return i


def autocorr_func_1d(x, norm=True):
    x = np.atleast_1d(x)
    if len(x.shape) != 1:
        raise ValueError("invalid dimensions for 1D autocorrelation function")
    n = next_pow_two(len(x))

    # Compute the FFT and then (from that) the auto-correlation function
    f = np.fft.fft(x - np.mean(x), n=2 * n)
    acf = np.fft.ifft(f * np.conjugate(f))[: len(x)].real
    acf /= 4 * n

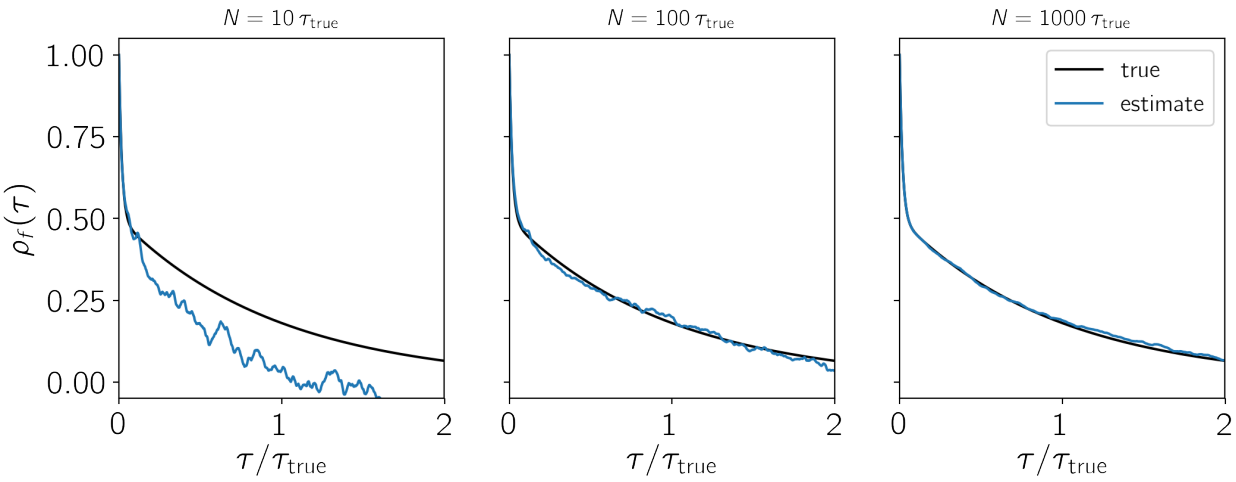
    # Optionally normalize
    if norm:
        acf /= acf[0]

    return acf


# Make plots of ACF estimate for a few different chain lengths
window = int(2 * true_tau)
tau = np.arange(window + 1)
f0 = kernel.get_value(tau) / kernel.get_value(0.0)

# Loop over chain lengths:
fig, axes = plt.subplots(1, 3, figsize=(12, 4), sharex=True, sharey=True)
for n, ax in zip([10, 100, 1000], axes):
    nn = int(true_tau * n)
    ax.plot(tau / true_tau, f0, "k", label="true")
    ax.plot(tau / true_tau, autocorr_func_1d(y[0, :nn])[: window + 1], label="estimate
↪")
    ax.set_title(r"$N = {0}\,\tau_\mathrm{{true}}$".format(n), fontsize=14)
    ax.set_xlabel(r"$\tau / \tau_\mathrm{true}$")

axes[0].set_ylabel(r"$\rho_f(\tau)$")
axes[-1].set_xlim(0, window / true_tau)
axes[-1].set_ylim(-0.05, 1.05)
axes[-1].legend(fontsize=14);
```

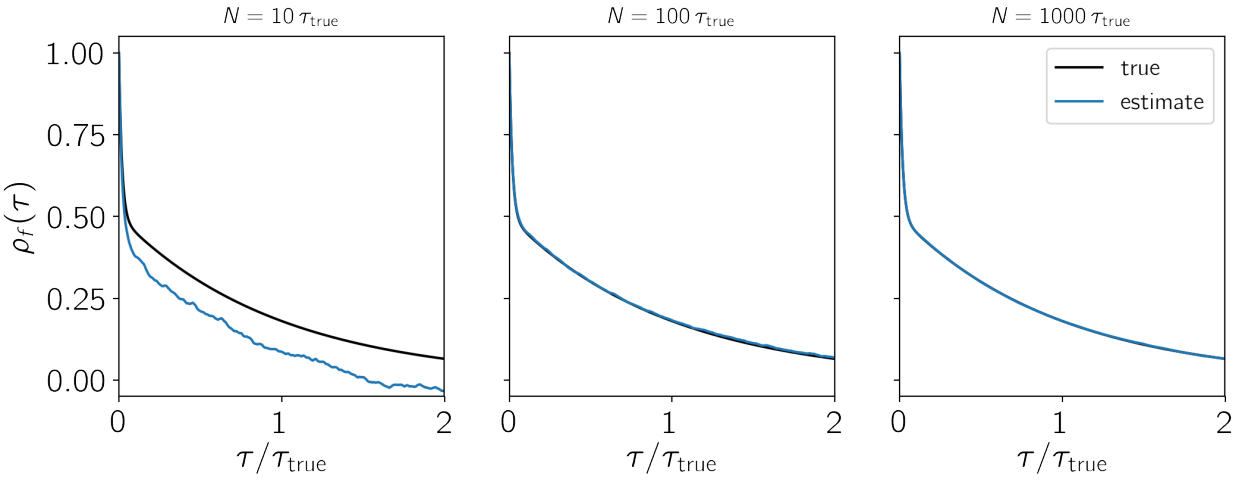


This figure shows how the empirical estimate of the normalized autocorrelation function changes as more samples are generated. In each panel, the true autocorrelation function is shown as a black curve and the empirical estimator is shown as a blue line.

Instead of estimating the autocorrelation function using a single chain, we can assume that each chain is sampled from the same stochastic process and average the estimate over ensemble members to reduce the variance. It turns out that we'll actually do this averaging later in the process below, but it can be useful to show the mean autocorrelation function for visualization purposes.

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4), sharex=True, sharey=True)
for n, ax in zip([10, 100, 1000], axes):
    nn = int(true_tau * n)
    ax.plot(tau / true_tau, f0, "k", label="true")
    f = np.mean(
        [autocorr_func_1d(y[i, :nn], norm=False)[: window + 1] for i in␣
↪range(len(y))],
        axis=0,
    )
    f /= f[0]
    ax.plot(tau / true_tau, f, label="estimate")
    ax.set_title(r"$N = {0}\,\tau_\mathrm{{true}}$".format(n), fontsize=14)
    ax.set_xlabel(r"$\tau / \tau_\mathrm{true}$")

axes[0].set_ylabel(r"$\rho_f(\tau)$")
axes[-1].set_xlim(0, window / true_tau)
axes[-1].set_ylim(-0.05, 1.05)
axes[-1].legend(fontsize=14);
```



Now let's estimate the autocorrelation time using these estimated autocorrelation functions. Goodman & Weare (2010) suggested averaging the ensemble over walkers and computing the autocorrelation function of the mean chain to lower the variance of the estimator and that was what was originally implemented in emcee. Since then, @fardal on GitHub suggested that other estimators might have lower variance. This is absolutely correct and, instead of the Goodman & Weare method, we now recommend computing the autocorrelation time for each walker (it's actually possible to still use the ensemble to choose the appropriate window) and then average these estimates.

Here is an implementation of each of these methods and a plot showing the convergence as a function of the chain length:

```
# Automated windowing procedure following Sokal (1989)
def auto_window(taus, c):
    m = np.arange(len(taus)) < c * taus
    if np.any(m):
        return np.argmin(m)
    return len(taus) - 1


# Following the suggestion from Goodman & Weare (2010)
def autocorr_gw2010(y, c=5.0):
    f = autocorr_func_1d(np.mean(y, axis=0))
    taus = 2.0 * np.cumsum(f) - 1.0
    window = auto_window(taus, c)
    return taus[window]


def autocorr_new(y, c=5.0):
    f = np.zeros(y.shape[1])
    for yy in y:
        f += autocorr_func_1d(yy)
    f /= len(y)
    taus = 2.0 * np.cumsum(f) - 1.0
    window = auto_window(taus, c)
    return taus[window]

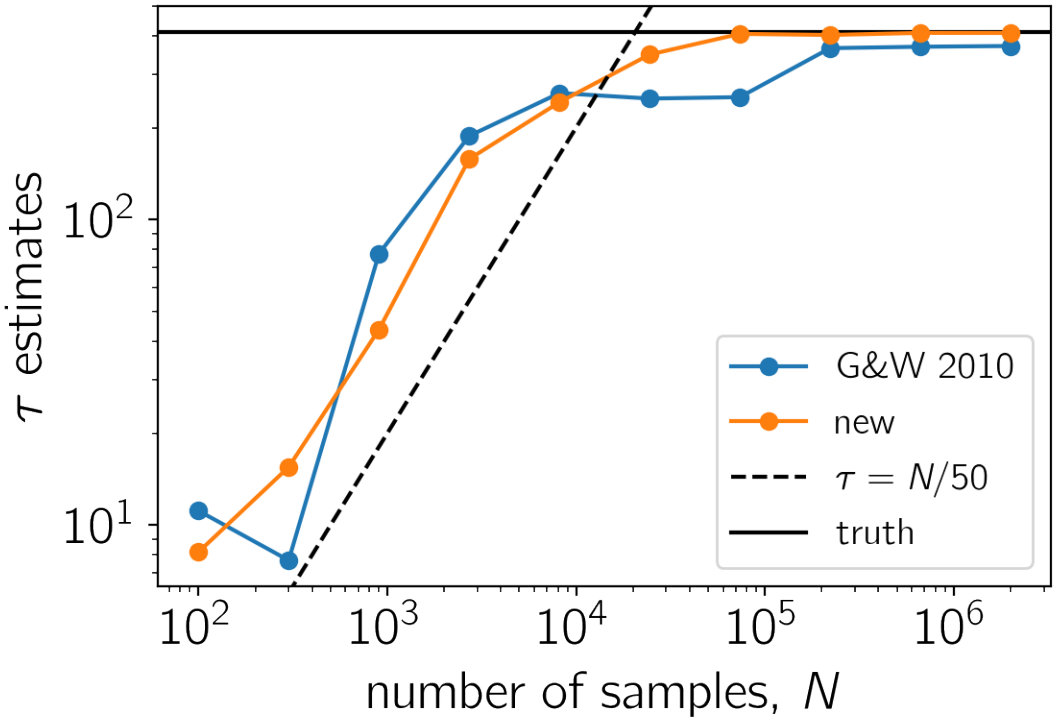

# Compute the estimators for a few different chain lengths
N = np.exp(np.linspace(np.log(100), np.log(y.shape[1]), 10)).astype(int)
gw2010 = np.empty(len(N))
new = np.empty(len(N))
```

(continues on next page)

(continued from previous page)

```
for i, n in enumerate(N):
    gw2010[i] = autocorr_gw2010(y[:, :n])
    new[i] = autocorr_new(y[:, :n])

# Plot the comparisons
plt.loglog(N, gw2010, "o-", label="G\&W 2010")
plt.loglog(N, new, "o-", label="new")
ylim = plt.gca().get_ylim()
plt.plot(N, N / 50.0, "--k", label=r"$\tau = N/50$")
plt.axhline(true_tau, color="k", label="truth", zorder=-100)
plt.ylim(ylim)
plt.xlabel("number of samples, $N$")
plt.ylabel(r"$\tau$ estimates")
plt.legend(fontsize=14);
```



In this figure, the true autocorrelation time is shown as a horizontal line and it should be clear that both estimators give outrageous results for the short chains. It should also be clear that the new algorithm has lower variance than the original method based on Goodman & Weare. In fact, even for moderately long chains, the old method can give dangerously over-confident estimates. For comparison, we have also plotted the $\tau = N/50$ line to show that, once the estimate crosses that line, The estimates are starting to get more reasonable. This suggests that you probably shouldn't trust any estimate of τ unless you have more than $F \times \tau$ samples for some $F \geq 50$. Larger values of F will be more conservative, but they will also (obviously) require longer chains.

2.12.4 A more realistic example

Now, let's run an actual Markov chain and test these methods using those samples. So that the sampling isn't completely trivial, we'll sample a multimodal density in three dimensions.

```
import emcee


def log_prob(p):
    return np.logaddexp(-0.5 * np.sum(p ** 2), -0.5 * np.sum((p - 4.0) ** 2))

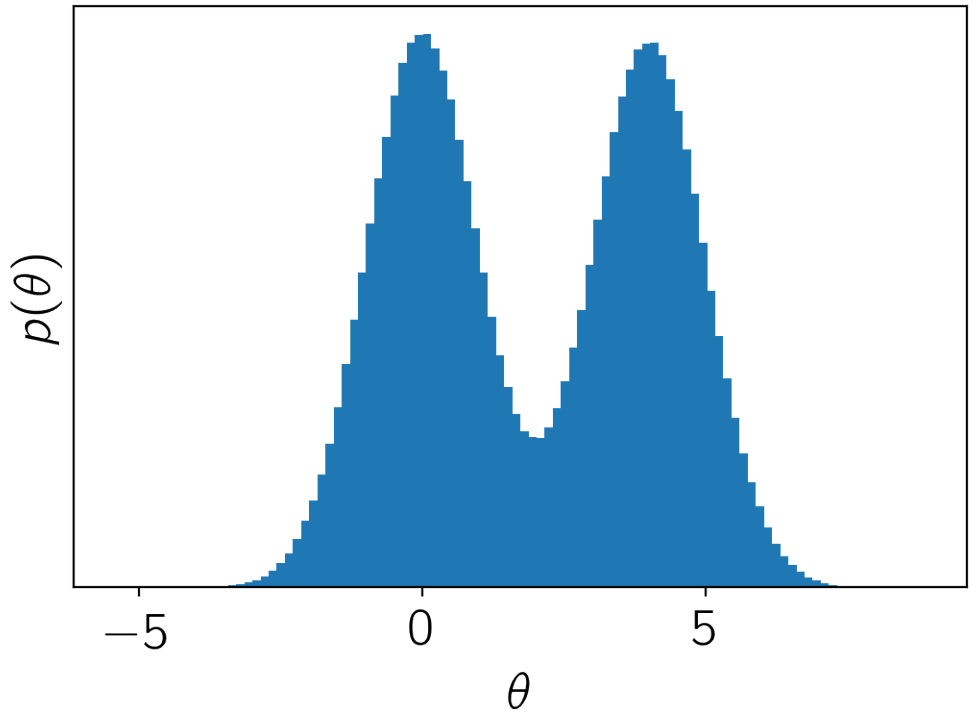

sampler = emcee.EnsembleSampler(32, 3, log_prob)
sampler.run_mcmc(
    np.concatenate((np.random.randn(16, 3), 4.0 + np.random.randn(16, 3)), axis=0),
    500000,
    progress=True,
);
```

```
100%|| 500000/500000 [07:18<00:00, 1139.29it/s]
```

Here's the marginalized density in the first dimension.

```
chain = sampler.get_chain()[:, :, 0].T

plt.hist(chain.flatten(), 100)
plt.gca().set_yticks([])
plt.xlabel(r"$\theta$")
plt.ylabel(r"$p(\theta)$");
```



And here's the comparison plot showing how the autocorrelation time estimates converge with longer chains.

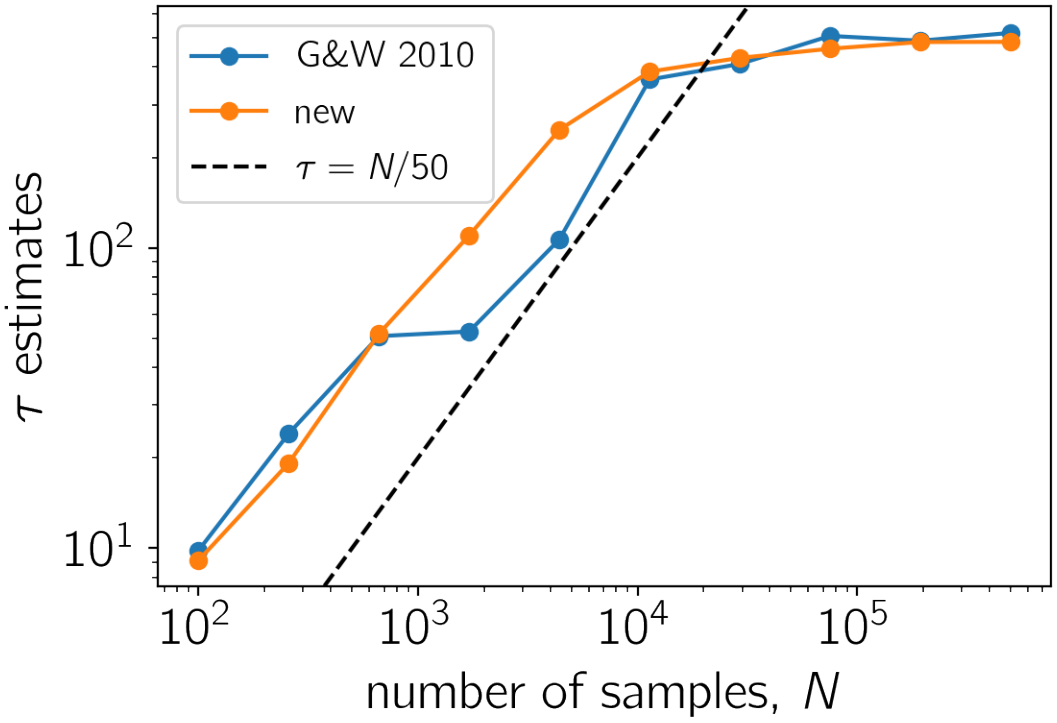
```
# Compute the estimators for a few different chain lengths
N = np.exp(np.linspace(np.log(100), np.log(chain.shape[1]), 10)).astype(int)
gw2010 = np.empty(len(N))
```

(continues on next page)

(continued from previous page)

```
new = np.empty(len(N))
for i, n in enumerate(N):
    gw2010[i] = autocorr_gw2010(chain[:, :n])
    new[i] = autocorr_new(chain[:, :n])

# Plot the comparisons
plt.loglog(N, gw2010, "o-", label="G\&W 2010")
plt.loglog(N, new, "o-", label="new")
ylim = plt.gca().get_ylim()
plt.plot(N, N / 50.0, "--k", label=r"$\tau = N/50$")
plt.ylim(ylim)
plt.xlabel("number of samples, $N$")
plt.ylabel(r"$\tau$ estimates")
plt.legend(fontsize=14);
```



As before, the short chains give absurd estimates of τ, but the new method converges faster and with lower variance than the old method. The $\tau = N/50$ line is also included as above as an indication of where we might start trusting the estimates.

2.12.5 What about shorter chains?

Sometimes it just might not be possible to run chains that are long enough to get a reliable estimate of τ using the methods described above. In these cases, you might be able to get an estimate using parametric models for the autocorrelation. One example would be to fit an autoregressive model to the chain and using that to estimate the autocorrelation time.

As an example, we'll use celerite to fit for the maximum likelihood autocorrelation function and then compute an estimate of τ based on that model. The celerite model that we're using is equivalent to a second-order ARMA model and it appears to be a good choice for this example, but we're not going to promise anything here about the general

applicability and we caution care whenever estimating autocorrelation times using short chains.

```
from scipy.optimize import minimize


def autocorr_ml(y, thin=1, c=5.0):
    # Compute the initial estimate of tau using the standard method
    init = autocorr_new(y, c=c)
    z = y[:, ::thin]
    N = z.shape[1]

    # Build the GP model
    tau = max(1.0, init / thin)
    kernel = terms.RealTerm(
        np.log(0.9 * np.var(z)), -np.log(tau), bounds=[(-5.0, 5.0), (-np.log(N), 0.0)]
    )
    kernel += terms.RealTerm(
        np.log(0.1 * np.var(z)),
        -np.log(0.5 * tau),
        bounds=[(-5.0, 5.0), (-np.log(N), 0.0)],
    )
    gp = celerite.GP(kernel, mean=np.mean(z))
    gp.compute(np.arange(z.shape[1]))

    # Define the objective
    def nll(p):
        # Update the GP model
        gp.set_parameter_vector(p)

        # Loop over the chains and compute likelihoods
        v, g = zip(*(gp.grad_log_likelihood(z0, quiet=True) for z0 in z))

        # Combine the datasets
        return -np.sum(v), -np.sum(g, axis=0)

    # Optimize the model
    p0 = gp.get_parameter_vector()
    bounds = gp.get_parameter_bounds()
    soln = minimize(nll, p0, jac=True, bounds=bounds)
    gp.set_parameter_vector(soln.x)

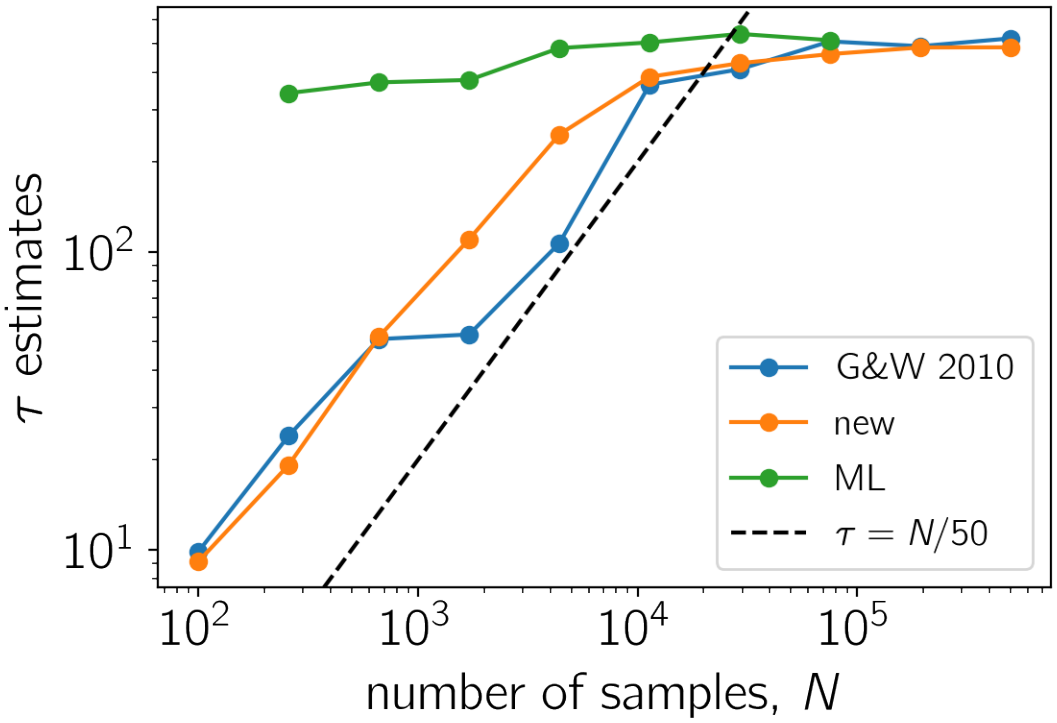
    # Compute the maximum likelihood tau
    a, c = kernel.coefficients[:2]
    tau = thin * 2 * np.sum(a / c) / np.sum(a)
    return tau


# Calculate the estimate for a set of different chain lengths
ml = np.empty(len(N))
ml[:] = np.nan
for j, n in enumerate(N[1:8]):
    i = j + 1
    thin = max(1, int(0.05 * new[i]))
    ml[i] = autocorr_ml(chain[:, :n], thin=thin)
```

```
/Users/dforeman/anaconda3/lib/python3.6/site-packages/autograd/numpy/numpy_
↪vjps.py:444: FutureWarning: Using a non-tuple sequence for multidimensional␣
↪indexing is deprecated; use arr[tuple(seq)] instead of arr[seq]. In the␣
```

```
↪future this will be interpreted as an array index, arr[np.array(seq)],␣
↪which will result either in an error or a different result.
  return lambda g: g[idxs]
```

```
# Plot the comparisons
plt.loglog(N, gw2010, "o-", label="G\&W 2010")
plt.loglog(N, new, "o-", label="new")
plt.loglog(N, ml, "o-", label="ML")
ylim = plt.gca().get_ylim()
plt.plot(N, N / 50.0, "--k", label=r"$\tau = N/50$")
plt.ylim(ylim)
plt.xlabel("number of samples, $N$")
plt.ylabel(r"$\tau$ estimates")
plt.legend(fontsize=14);
```



This figure is the same as the previous one, but we've added the maximum likelihood estimates for τ in green. In this case, this estimate seems to be robust even for very short chains with $N \sim \tau$. **Note:** This tutorial was generated from an IPython notebook that can be downloaded here.

2.13 Saving & monitoring 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.

In this demo, we will demonstrate how you can use the new `backends.HDFBackend` to save your results to a HDF5 file as the chain runs. To execute this, you'll first need to install the h5py library.

We'll also monitor the autocorrelation time at regular intervals (see *Autocorrelation analysis & convergence*) to judge convergence.

We will set up the problem as usual with one small change:

```
import emcee
import numpy as np

np.random.seed(42)

# The definition of the log probability function
# We'll also use the "blobs" feature to track the "log prior" for each step
def log_prob(theta):
    log_prior = -0.5 * np.sum((theta - 1.0) ** 2 / 100.0)
    log_prob = -0.5 * np.sum(theta ** 2) + log_prior
    return log_prob, log_prior


# Initialize the walkers
coords = np.random.randn(32, 5)
nwalkers, ndim = coords.shape

# Set up the backend
# Don't forget to clear it in case the file already exists
filename = "tutorial.h5"
backend = emcee.backends.HDFBackend(filename)
backend.reset(nwalkers, ndim)

# Initialize the sampler
sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob, backend=backend)
```

The difference here was the addition of a "backend". This choice will save the samples to a file called `tutorial.h5` in the current directory. Now, we'll run the chain for up to 10,000 steps and check the autocorrelation time every 100 steps. If the chain is longer than 100 times the estimated autocorrelation time and if this estimate changed by less than 1%, we'll consider things converged.

```
max_n = 100000

# We'll track how the average autocorrelation time estimate changes
index = 0
autocorr = np.empty(max_n)

# This will be useful to testing convergence
old_tau = np.inf

# Now we'll sample for up to max_n steps
for sample in sampler.sample(coords, iterations=max_n, progress=True):
    # Only check convergence every 100 steps
    if sampler.iteration % 100:
        continue

    # Compute the autocorrelation time so far
    # Using tol=0 means that we'll always get an estimate even
    # if it isn't trustworthy
    tau = sampler.get_autocorr_time(tol=0)
    autocorr[index] = np.mean(tau)
    index += 1

    # Check convergence
    converged = np.all(tau * 100 < sampler.iteration)
    converged &= np.all(np.abs(old_tau - tau) / tau < 0.01)
```

(continues on next page)

(continued from previous page)

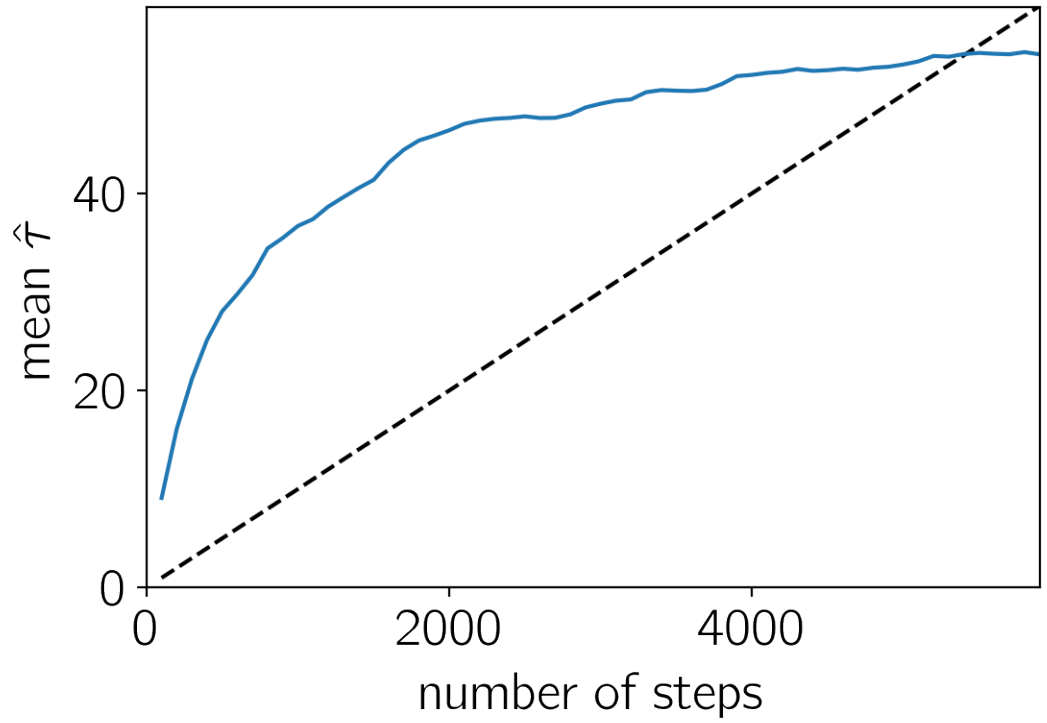
```
    if converged:
        break
    old_tau = tau
```

```
6%|         | 5900/100000 [00:56<14:59, 104.58it/s]
```

Now let's take a look at how the autocorrelation time estimate (averaged across dimensions) changed over the course of this run. In this plot, the τ estimate is plotted (in blue) as a function of chain length and, for comparison, the $N > 100\,\tau$ threshold is plotted as a dashed line.

```
import matplotlib.pyplot as plt

n = 100 * np.arange(1, index + 1)
y = autocorr[:index]
plt.plot(n, n / 100.0, "--k")
plt.plot(n, y)
plt.xlim(0, n.max())
plt.ylim(0, y.max() + 0.1 * (y.max() - y.min()))
plt.xlabel("number of steps")
plt.ylabel(r"mean $\hat{\tau}$");
```



As usual, we can also access all the properties of the chain:

```
import corner

tau = sampler.get_autocorr_time()
burnin = int(2 * np.max(tau))
thin = int(0.5 * np.min(tau))
samples = sampler.get_chain(discard=burnin, flat=True, thin=thin)
log_prob_samples = sampler.get_log_prob(discard=burnin, flat=True, thin=thin)
```

(continues on next page)

(continued from previous page)

```
log_prior_samples = sampler.get_blobs(discard=burnin, flat=True, thin=thin)

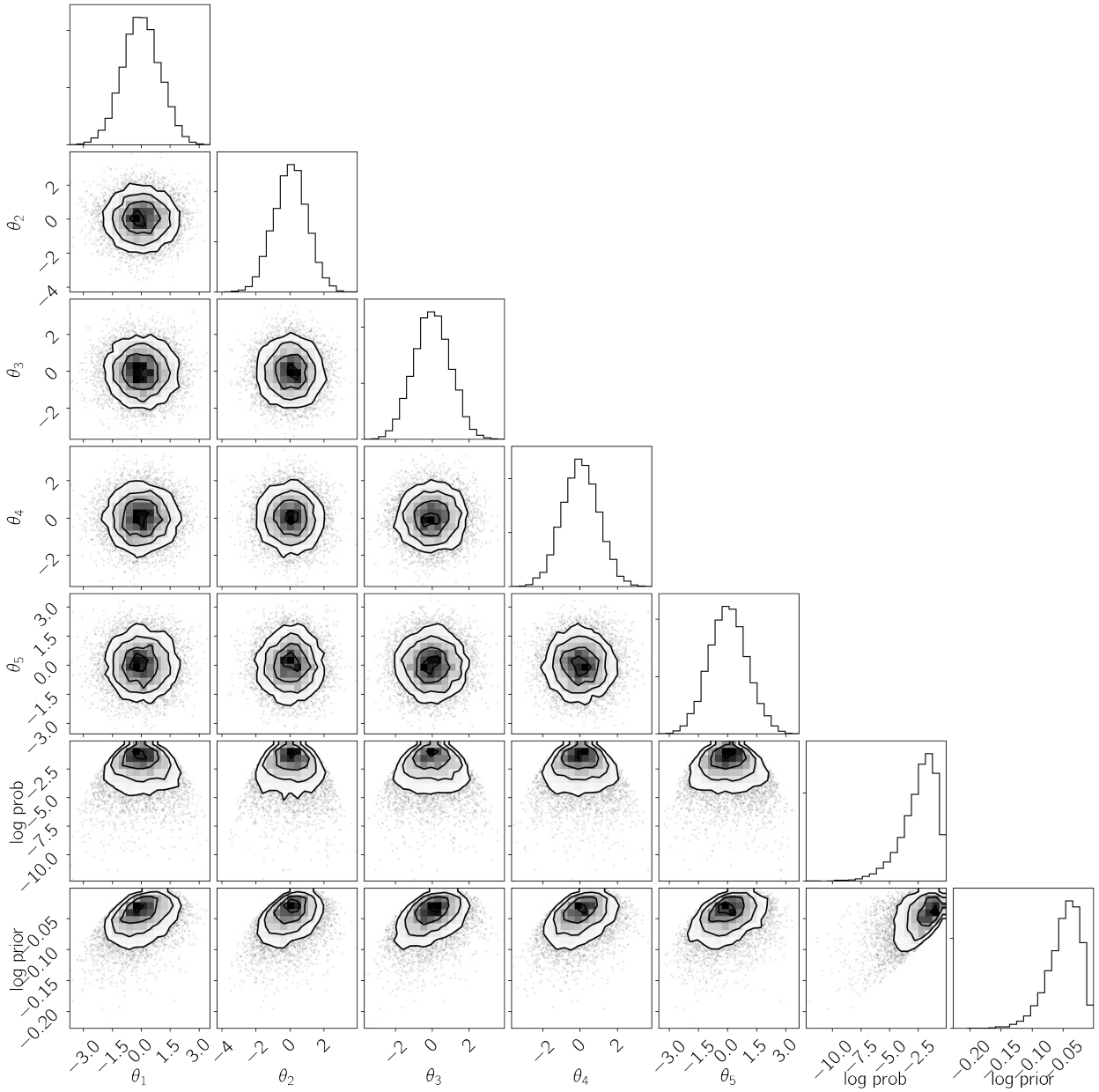
print("burn-in: {0}".format(burnin))
print("thin: {0}".format(thin))
print("flat chain shape: {0}".format(samples.shape))
print("flat log prob shape: {0}".format(log_prob_samples.shape))
print("flat log prior shape: {0}".format(log_prior_samples.shape))

all_samples = np.concatenate(
    (samples, log_prob_samples[:, None], log_prior_samples[:, None]), axis=1
)

labels = list(map(r"$\theta_{{{0}}}$".format, range(1, ndim + 1)))
labels += ["log prob", "log prior"]

corner.corner(all_samples, labels=labels);
```

```
burn-in: 117
thin: 24
flat chain shape: (7680, 5)
flat log prob shape: (7680,)
flat log prior shape: (7680,)
```



But, since you saved your samples to a file, you can also open them after the fact using the *backends.HDFBackend*:

```
reader = emcee.backends.HDFBackend(filename)

tau = reader.get_autocorr_time()
burnin = int(2 * np.max(tau))
thin = int(0.5 * np.min(tau))
samples = reader.get_chain(discard=burnin, flat=True, thin=thin)
log_prob_samples = reader.get_log_prob(discard=burnin, flat=True, thin=thin)
log_prior_samples = reader.get_blobs(discard=burnin, flat=True, thin=thin)

print("burn-in: {0}".format(burnin))
print("thin: {0}".format(thin))
```

(continues on next page)

(continued from previous page)

```
print("flat chain shape: {0}".format(samples.shape))
print("flat log prob shape: {0}".format(log_prob_samples.shape))
print("flat log prior shape: {0}".format(log_prior_samples.shape))
```

```
burn-in: 117
thin: 24
flat chain shape: (7680, 5)
flat log prob shape: (7680,)
flat log prior shape: (7680,)
```

This should give the same output as the previous code block, but you'll notice that there was no reference to `sampler` here at all.

If you want to restart from the last sample, you can just leave out the call to *backends.HDFBackend.reset()*:

```
new_backend = emcee.backends.HDFBackend(filename)
print("Initial size: {0}".format(new_backend.iteration))
new_sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob, backend=new_backend)
new_sampler.run_mcmc(None, 100)
print("Final size: {0}".format(new_backend.iteration))
```

```
Initial size: 5900
Final size: 6000
```

If you want to save *additional* `emcee` runs, you can do so on the same file as long as you set the `name` of the backend object to something other than the default:

```
run2_backend = emcee.backends.HDFBackend(filename, name="mcmc_second_prior")

# this time, with a subtly different prior
def log_prob2(theta):
    log_prior = -0.5 * np.sum((theta - 2.0) ** 2 / 100.0)
    log_prob = -0.5 * np.sum(theta ** 2) + log_prior
    return log_prob, log_prior


# Rinse, Wash, and Repeat as above
coords = np.random.randn(32, 5)
nwalkers, ndim = coords.shape
sampler2 = emcee.EnsembleSampler(nwalkers, ndim, log_prob2, backend=run2_backend)

# note: this is *not* necessarily the right number of iterations for this
# new prior.  But it will suffice  to demonstrate the second backend.
sampler2.run_mcmc(coords, new_backend.iteration, progress=True);
```

```
100%|| 6000/6000 [00:49<00:00, 122.13it/s]
```

And now you can see *both* runs are in the file:

```
import h5py

with h5py.File(filename, "r") as f:
    print(list(f.keys()))
```

```
['mcmc', 'mcmc_second_prior']
```

CHAPTER 3

License & Attribution

Copyright 2010-2019 Dan Foreman-Mackey and contributors.

emcee is free software made available under the MIT License. For details see the `LICENSE`.

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 4

Changelog

4.1 3.0.1 (2010-10-28)

- Added support for long double dtypes
- Prepared manuscript to submit to JOSS
- Improved packaging and release infrastructure
- Fixed bug in initial linear dependence test

4.2 3.0.0 (2019-09-30)

- Added progress bars using tqdm.
- Added HDF5 backend using h5py.
- Added new `Move` interface for more flexible specification of proposals.
- Improved autocorrelation time estimation algorithm.
- Switched documentation to using Jupyter notebooks for tutorials.
- More details can be found on the docs.

4.3 2.2.0 (2016-07-12)

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

4.4 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`.

4.5 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`.

4.6 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.

4.7 1.1.3 (2012-11-22)

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

4.8 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.

4.9 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.

4.10 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.

4.11 1.0.1 (2012-03-31)

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

4.12 1.0.0 (2012-02-15)

- Initial release.

Python Module Index

e

emcee, 17

Index

A

acceptance_fraction (*emcee.EnsembleSampler attribute*), 6

B

Backend (*class in emcee.backends*), 13

C

compute_log_prob() (*emcee.EnsembleSampler method*), 6

D

DEMove (*class in emcee.moves*), 10
DESnookerMove (*class in emcee.moves*), 10

E

emcee (*module*), 6, 9, 11, 13, 17, 18, 21, 29, 34, 44
EnsembleSampler (*class in emcee*), 6

F

function_1d() (*in module emcee.autocorr*), 17

G

GaussianMove (*class in emcee.moves*), 11
get_autocorr_time() (*emcee.backends.Backend method*), 13
get_autocorr_time() (*emcee.backends.HDFBackend method*), 15
get_autocorr_time() (*emcee.EnsembleSampler method*), 7
get_blobs() (*emcee.backends.Backend method*), 14
get_blobs() (*emcee.backends.HDFBackend method*), 15
get_blobs() (*emcee.EnsembleSampler method*), 7
get_chain() (*emcee.backends.Backend method*), 14
get_chain() (*emcee.backends.HDFBackend method*), 16
get_chain() (*emcee.EnsembleSampler method*), 7
get_last_sample() (*emcee.backends.Backend method*), 14
get_last_sample() (*emcee.backends.HDFBackend method*), 16
get_last_sample() (*emcee.EnsembleSampler method*), 7
get_log_prob() (*emcee.backends.Backend method*), 14
get_log_prob() (*emcee.backends.HDFBackend method*), 16
get_log_prob() (*emcee.EnsembleSampler method*), 7
grow() (*emcee.backends.Backend method*), 14
grow() (*emcee.backends.HDFBackend method*), 16

H

has_blobs() (*emcee.backends.Backend method*), 15
has_blobs() (*emcee.backends.HDFBackend method*), 16
HDFBackend (*class in emcee.backends*), 15

I

integrated_time() (*in module emcee.autocorr*), 17

K

KDEMove (*class in emcee.moves*), 10

M

MHMove (*class in emcee.moves*), 11

P

propose() (*emcee.moves.MHMove method*), 11
propose() (*emcee.moves.RedBlueMove method*), 10

R

random_state (*emcee.EnsembleSampler attribute*), 8
RedBlueMove (*class in emcee.moves*), 9
reset() (*emcee.backends.Backend method*), 15
reset() (*emcee.backends.HDFBackend method*), 16

reset() (*emcee.EnsembleSampler method*), 8
run_mcmc() (*emcee.EnsembleSampler method*), 8

S

sample() (*emcee.EnsembleSampler method*), 8
save_step() (*emcee.backends.Backend method*), 15
save_step() (*emcee.backends.HDFBackend method*), 17
shape (*emcee.backends.Backend attribute*), 15
shape (*emcee.backends.HDFBackend attribute*), 17
State (*class in emcee*), 9
StretchMove (*class in emcee.moves*), 10

W

WalkMove (*class in emcee.moves*), 10