zeus

Release 2.4.1

Jan 12, 2023

Cookbook Recipes

zeus is a Python implementation of the Ensemble Slice Sampling method.

- Fast & Robust *Bayesian Inference*,
- Efficient *Markov Chain Monte Carlo (MCMC)*,
- Black-box inference, no hand-tuning,
- Excellent performance in terms of autocorrelation time and convergence rate,
- Scale to multiple CPUs without any extra effort,
- Automated Convergence diagnostics. NEW

For instance, if you wanted to draw samples from a *10-dimensional Normal distribution*, you would do something like:

```
import zeus
import numpy as np

def log_prob(x, ivar):
    return - 0.5 * np.sum(ivar * x**2.0)

nsteps, nwalkers, ndim = 1000, 100, 10
ivar = 1.0 / np.random.rand(ndim)
start = np.random.randn(nwalkers, ndim)

sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob, args=[ivar])
sampler.run_mcmc(start, nsteps)
chain = sampler.get_chain(flat=True)
```

CHAPTER 1

Installation

To install `zeus` using `pip` run:

```
pip install zeus-mcmc
```

To install `zeus` in a [Ana]Conda environment use:

```
conda install -c conda-forge zeus-mcmc
```

CHAPTER 2

Getting Started

- See the *Cookbook* page to learn how to perform Bayesian Inference using `zeus`.
- See the *Frequently Asked Questions* page for frequently asked questions about `zeus`' operation.
- See the *API Reference* page for detailed API documentation.

CHAPTER 3

Citation

Please cite the following papers if you found this code useful in your research:

```
@article{karamanis2021zeus,
        title={zeus: A Python implementation of Ensemble Slice Sampling for
↪efficient Bayesian parameter inference},
        author={Karamanis, Minas and Beutler, Florian and Peacock, John A},
        journal={arXiv preprint arXiv:2105.03468},
        year={2021}
       }

@article{karamanis2020ensemble,
        title = {Ensemble slice sampling: Parallel, black-box and gradient-free
↪inference for correlated & multimodal distributions},
        author = {Karamanis, Minas and Beutler, Florian},
        journal = {arXiv preprint arXiv: 2002.06212},
        year = {2020}
       }
```

CHAPTER 4

Licence

CHAPTER 5

Changelog

2.4.1 (17/11/21)

- Introduced `ParallelSplitRCallback` callback function for checking Gelman-Rubin statistics during `MPI` runs.

2.4.0 (01/11/21)

- Introduced callback interface.
- Added convergence diagnostics.
- Added `H5DF` support.

2.3.1 (03/08/21)

- Raise exception if model fails.

2.3.0 (25/02/21)

- Added `sample` method which advances the chain as a generator.
- Added `light_mode`. When used, `light_mode` can significantly reduce the number of log likelihood evaluations and increase the general efficiency of the algorithm. `light_mode` works by performing no expansions after the end of the tuning phase. The scale factor is set to its opttimal value. This works best for approximately Gaussian distributions.
- Added `start=None` support for `run_mcmc`. When used, the sampler proceeds from the last known position of the walkers.
- Added support for both `thin` and `thin_by` arguments.

2.2.2 (21/02/21)

- Added `log_prob0` and `blobs0` arguments in `run`.
- Added `get_last_sample()`, `get_last_log_prob()` and `get_last_blobs()` methods.

2.2.0 (03/11/20)

- Improved vectorization.

2.1.1 (29/10/20)

- Added `blobs` interface to track arbitrary metadata.
- Updated `GlobalMove` and multimodal example.
- Fixed minor bugs.

2.0.0 (05/10/20)

- Added new `Moves` interface (e.g. `DifferentialMove`, `GlobalMove`, etc).
- Plotting capabilities (i.e. `cornerplot`).
- Updated docs.
- Fixed minor bugs.

1.2.2 (19/09/20)

- `Sampler` class is deprecated. New `EnsembleSampler` class in now available.
- New estimator for the Integrated Autocorrelation Time. It's accurate even with short chains.
- Updated `ChainManager` to handle thousands of CPUs.

1.2.1 (04/08/20)

- Changed to Flat-not-nested philosophy for diagnostics and `ChainManager`.

1.2.0 (03/08/20)

- Extended `ChainManager` with `gather`, `scatter`, and `bcast` tools.

1.1.0 (02/08/20)

- Added `ChainManager` to deploy into supercomputing clusters, parallelizing both chains and walkers.
- Added Convergence diagnostic tools (Gelman-Rubin, Geweke).

1.0.7 (11/05/20)

- Improved parallel distribution of tasks

5.1 Cookbook

5.1.1 MCMC Sampling recipes

- ***Sampling from a multivariate Normal distribution*** Demonstrates how to sample from a correlated multivariate Gaussian distribution and how to perform the post-processing of the samples.
- ***Fitting a model to data*** In this recipe we are going to produce some mock data and use them to illustrate how *zeus* works in realistic scenarios.
- ***Sampling from multimodal distributions*** In this recipe we will demonstrate how one can use `zeus` with the `Moves` interface to sample efficiently from challenging high-dimensional multimodal distributions.

5.1.2 Parallelisation recipes

- ***Multiprocessing*** Use many CPUs to sample from an expensive-to-evaluate probability distribution even faster.
- ***MPI and ChainManager*** Distribute calculation to huge computer clusters.

5.1.3 Convergence Diagnostics and Saving Progress recipes NEW

- ***Automated Convergence Diagnostics using the callback interface*** **NEW** In this recipe we are going to use the callback interface to monitor convergence and stop sampling automatically.
- ***Saving progress to disk using h5py*** **NEW** In this recipe we are going to use the callback interface to save the samples and their corresponding log-probability values in a `.h5` file.
- ***Parallel sampling using MPI and Gelman-Rubin convergence diagnostics*** **NEW** In this recipe we are going to use the ChainManager to run zeus in parallel using MPI and terminate sampling automatically using Gelman-Rubin diagnostics.
- ***Tracking metadata using the blobs interface*** We introduce the blobs interface. An easy way for the user to track arbitrary metadata for every sample of the chain.

Sampling from a multivariate Normal distribution

Lets import some libraries that we're going to use.

We're going to need:

- *numpy* because there's nothing we can do without it,
- *sklearn* to produce a mock covariance matrix for the normal distribution,
- *matplotlib* to plot the covariance matrix and visually inspect our results,

and of course **zeus** to perform MCMC Bayesian Inference.

[1]:
```
import numpy as np
from sklearn.datasets import make_spd_matrix
import matplotlib.pyplot as plt
%matplotlib inline
import zeus
```

Now we need to define:

- *ndim* the number of dimensions/parameters of our distribution,
- *nwalkers* the number of walkers, as a rule of thumb we choose the minimum value, twice the number of parameters,
- *nsteps* the numper of steps/generations.

We also want to:

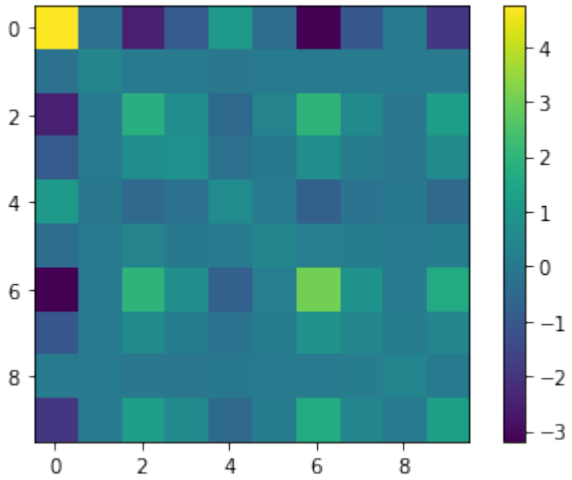
- produce a mock covariance matrix using the *make_spd_matrix* function of *scikit-learn*,
- compute its inverse,
- define a random mean vector for our posterior distribution,
- define the log probability of the posterior distribution as a python function,
- provide a starting point for the sampler.

[2]:
```
ndim = 10
nwalkers = 30
nsteps= 5000

C = make_spd_matrix(ndim)
plt.imshow(C)
```

(continues on next page)

(continued from previous page)

```
plt.colorbar()
plt.show()

icov = np.linalg.inv(C)

mu = np.random.rand(ndim)

def log_prob(x, mu, icov):
    return -0.5 * np.dot(np.dot((x-mu).T,icov),(x-mu))

start = np.random.randn(nwalkers, ndim)
```



Now we are ready to do some inference.

- First we initialise the sampler by calling the *zeus.EnsembleSampler* class,
- and then we run the MCMC by calling the *run_mcmc* method.

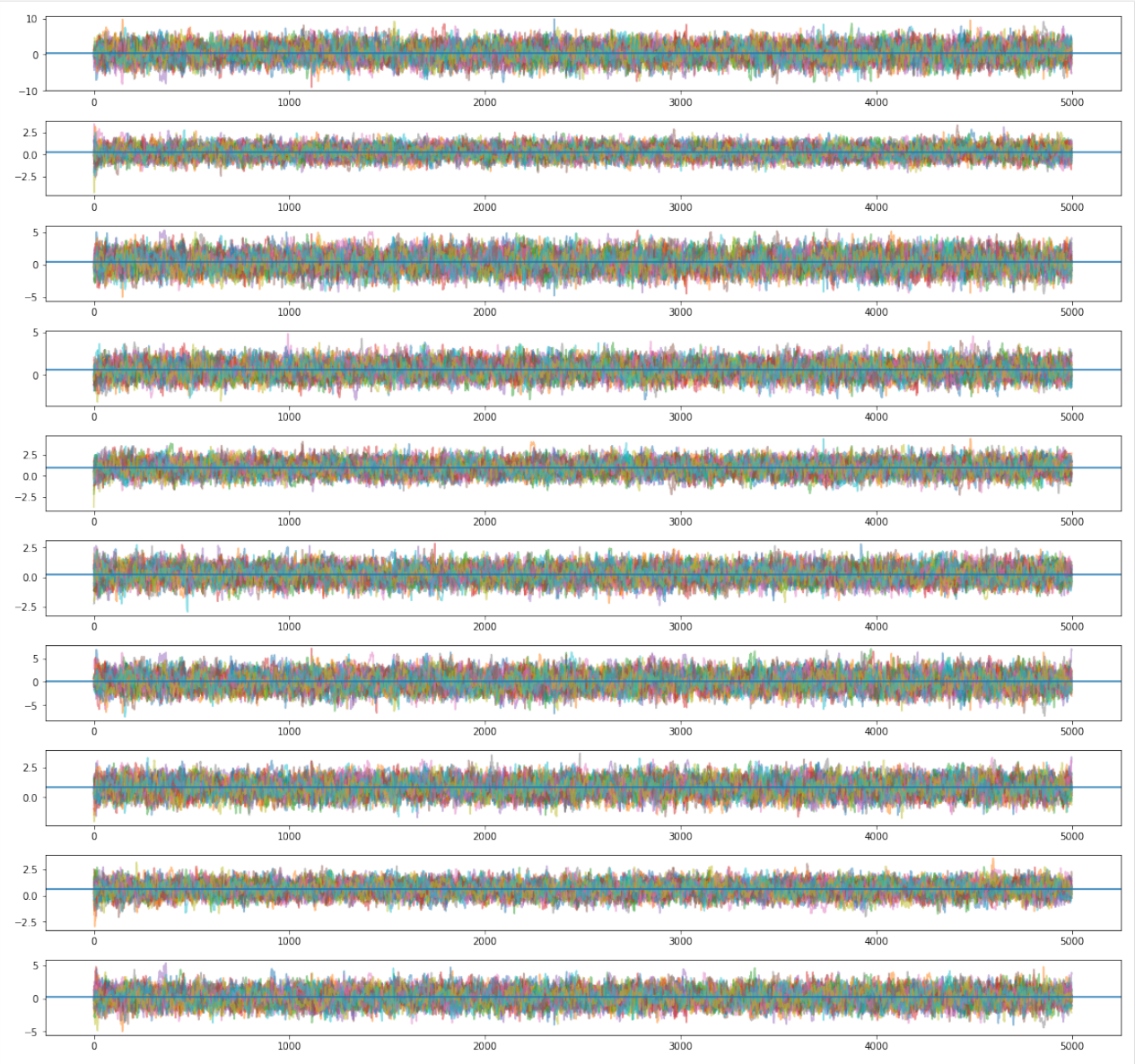
This is going to be very fast.

[3]:
```
sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob, args=[mu, icov])
sampler.run_mcmc(start, nsteps)
```

```
Initialising ensemble of 30 walkers...
Sampling progress : 100%|| 5000/5000 [00:22<00:00, 220.01it/s, nexp=0.9, ncon=0.8]
```

Alright, lets plot our chains to see what we've got. We can get the chains using the *sampler.get_chain()* method, their shape is (nsteps, nwalkers, ndim). So we want to iterate over all dimensions and plot the results. We also plot the true values of the parameters so that we can compare the results.

[4]:
```
plt.figure(figsize=(16,1.5*ndim))
for n in range(ndim):
    plt.subplot2grid((ndim, 1), (n, 0))
    plt.plot(sampler.get_chain()[:,:,n],alpha=0.5)
    plt.axhline(y=mu[n])
plt.tight_layout()
plt.show()
```



Great! This looks very good.

Now lets cut this burn-in phase off. We can either do this manually using *numpy* or even better use *zeus*'s *get_chain()* method. We are going to discard (or burn) the first half of the chain.

[5]:
```
chain = sampler.get_chain(flat=True, discard=2500)
```

We can now compute some usefull statistics:

[6]:
```
print('Percentiles')
print (np.percentile(chain, [16, 50, 84], axis=0))
print('Mean')
print (np.mean(chain, axis=0))
print('Standard Deviation')
print (np.std(chain, axis=0))
```

```
Percentiles
```

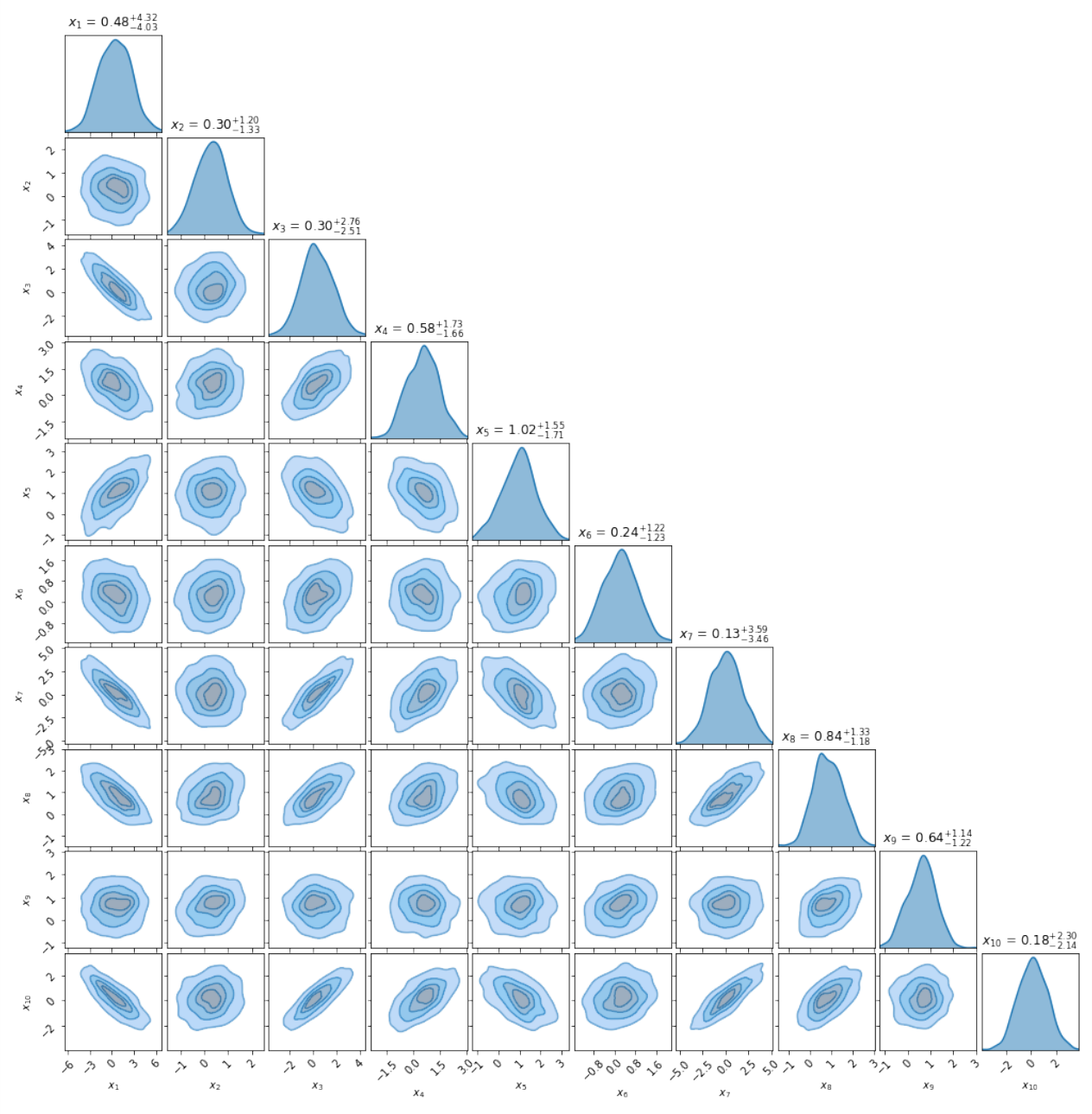
(continues on next page)

(continued from previous page)

```
[[-1.72926164 -0.37047414 -0.97883278 -0.27960606  0.19628294 -0.3995533
  -1.67860663  0.15444588 -0.0238074  -0.95754369]
 [ 0.49836064  0.27642868  0.36344295  0.59783351  1.00279701  0.23917056
   0.04884439  0.83016748  0.61841912  0.17759142]
 [ 2.65831953  0.9327647   1.71730444  1.45862505  1.76876848  0.89110814
   1.8100036   1.51475414  1.24719381  1.32263236]]
Mean
[0.47602812 0.28040841 0.37257773 0.59298147 0.98609419 0.24318362
 0.0637578  0.83432842 0.61438382 0.18063718]
Standard Deviation
[2.19336201 0.65521987 1.34839027 0.87488145 0.79438601 0.64576472
 1.7512727  0.6813218  0.63916386 1.14306749]
```

Last but not least, we can also plot the marginalised posterior distributions:

[7]:
```
fig, axes = zeus.cornerplot(chain[::100], size=(16,16))
```



Fitting a model to data

In this recipe we will demonstrate how to fit a simple model, namely a line, to some data. Although this example is simple, it illustrates what is the proper way of fitting our models to data and infering the parameters of the models.

Let us first import the main packages that we will use:

[1]:
```
# show plots inline in the notebook
%matplotlib inline

import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt

from IPython.display import display, Math

import zeus
```

The generative probabilistic model

In order to create our *synthetic* data we need to construct a *generative probabilistic model*.

We start by defining the *straight line* model and also setting the *true values* of the model parameters:

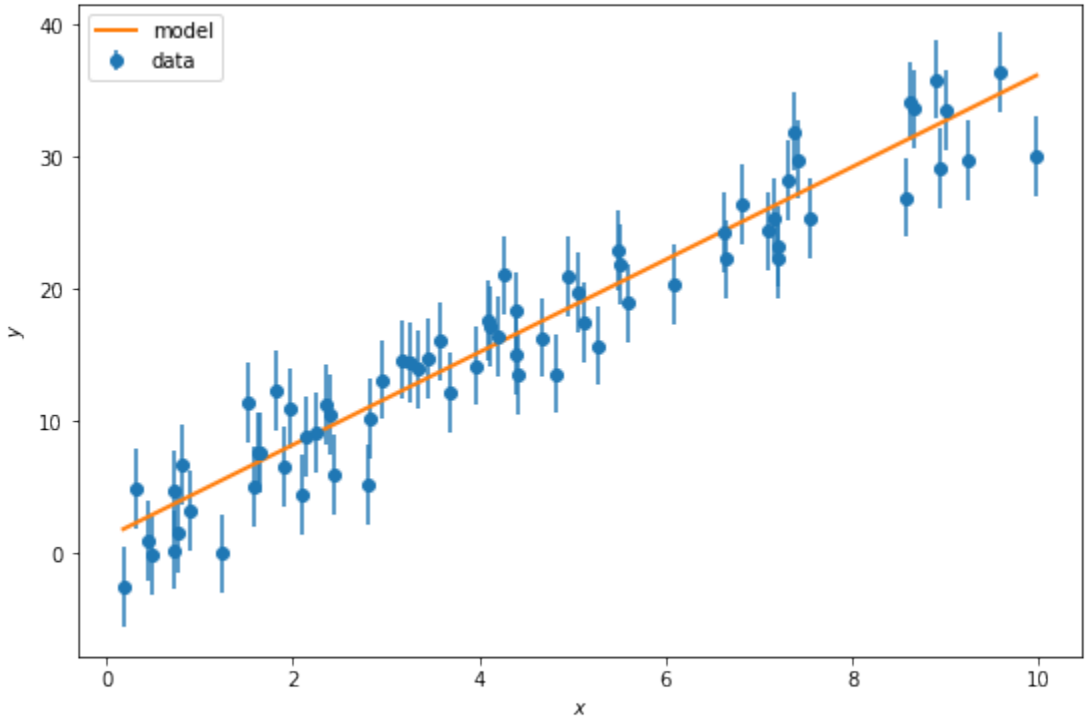
[2]:
```
# define the model function
def straight_line(x, m, c):
    ''' A straight line model: y = m*x + c '''
    return m*x + c

# set the true values of the model parameters for creating the data
m_true = 3.5 # gradient of the line
c_true = 1.2 # y-intercept of the line

# Set the x-coordinates of the data points
M = 70 # Number of data points
x = np.sort(10.0 * np.random.rand(M)) # their x-coordinates
```

We are now ready to generate the synthetic data. To this end, we evaluate the model function at the *true values* of *m (slope)* and *c (y-intercept)* and we add some random *Gaussian* noise of known amplitude *sigma*.

[3]:
```
# create the data - the model plus Gaussian noise
sigma = 3.0 # standard deviation of the noise
data = straight_line(x, m_true, c_true) + sigma * np.random.randn(M)
```

We can also plot the generative model and the data:

[4]:
```
plt.figure(figsize=(9,6))
plt.errorbar(x, data, yerr=sigma, fmt="o", label='data')
plt.plot(x, straight_line(x, m_true, c_true), '-', lw=2, label='model')
plt.legend()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.show()
```



The likelihood, prior, and posterior distributions

The first step to solve a problem is generally to write down the prior and likelihood functions. An important benefit of MCMC is that none of these probability densities need to be normalised.

Here we'll start with the natural logarithm of the prior probability:

[5]:
```
def logprior(theta):
    ''' The natural logarithm of the prior probability. '''

    lp = 0.

    # unpack the model parameters from the tuple
    m, c = theta

    # uniform prior on c
    cmin = -10. # lower range of prior
    cmax = 10.  # upper range of prior

    # set prior to 1 (log prior to 0) if in the range and zero (-inf) outside the␣
↪range
    lp = 0. if cmin < c < cmax else -np.inf

    # Gaussian prior on m
    mmu = 3.     # mean of the Gaussian prior
    msigma = 10. # standard deviation of the Gaussian prior
    lp -= 0.5*((m - mmu)/msigma)**2

    return lp
```

We assume that the likelihood is *Gaussian (Normal)*:

[6]:
```
def loglike(theta, data, sigma, x):
    '''The natural logarithm of the likelihood.'''

    # unpack the model parameters
    m, c = theta

    # evaluate the model
    md = straight_line(x, m, c)

    # return the log likelihood
    return -0.5 * np.sum(((md - data)/sigma)**2)
```

The log posterior is just the sum of the log prior and the log likelihood probability density functions:

[7]:
```
def logpost(theta, data, sigma, x):
    '''The natural logarithm of the posterior.'''

    return logprior(theta) + loglike(theta, data, sigma, x)
```

Sampling the posterior using *zeus*

We initialize and run zeus to sample from the posterior distribution. Thin only takes a few lines of code.

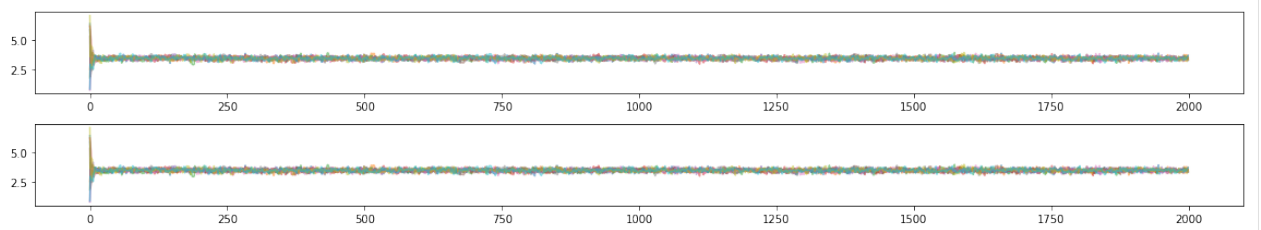
[8]:
```
ndim = 2 # Number of parameters/dimensions (e.g. m and c)
nwalkers = 10 # Number of walkers to use. It should be at least twice the number of␣
↪dimensions.
nsteps = 2000 # Number of steps/iterations.

start = 0.01 * np.random.randn(nwalkers, ndim) # Initial positions of the walkers.

sampler = zeus.EnsembleSampler(nwalkers, ndim, logpost, args=[data, sigma, x]) #␣
↪Initialise the sampler
sampler.run_mcmc(start, nsteps) # Run sampling
sampler.summary # Print summary diagnostics
```

```
Initialising ensemble of 10 walkers...
Sampling progress : 100%|| 2000/2000 [00:08<00:00, 237.71it/s, nexp=0.8, ncon=1.4]
Summary
-------
Number of Generations: 2000
Number of Parameters: 2
Number of Walkers: 10
Number of Tuning Generations: 24
Scale Factor: 3.03521
Mean Integrated Autocorrelation Time: 3.02
Effective Sample Size: 6629.56
Number of Log Probability Evaluations: 104165
Effective Samples per Log Probability Evaluation: 0.063645
```

Results

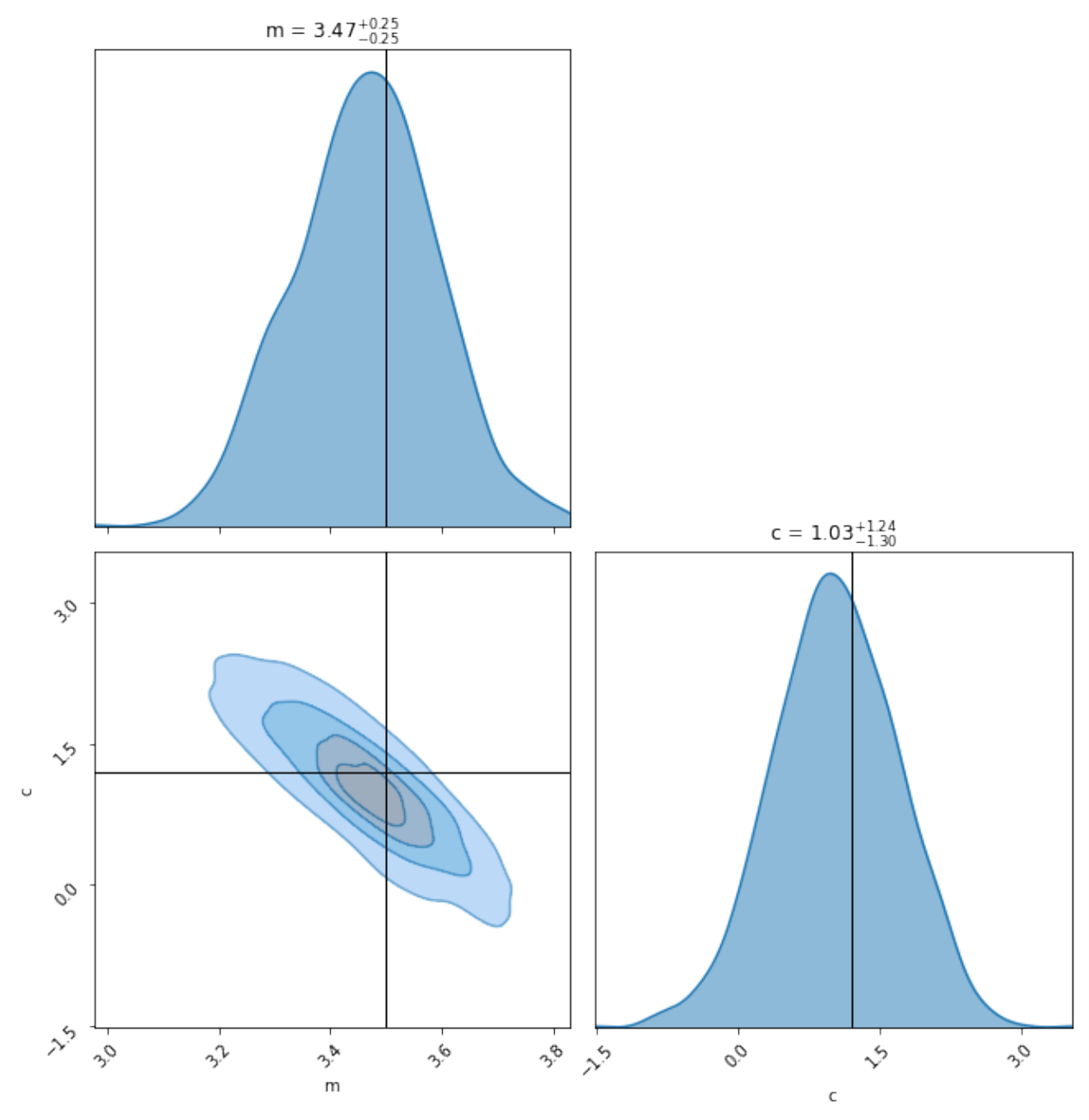
Lets plot the chains. We can see that the burn-in phase is very brief.

[9]:
```
plt.figure(figsize=(16,1.5*ndim))
for n in range(ndim):
    plt.subplot2grid((ndim, 1), (n, 0))
    plt.plot(sampler.get_chain()[:,:,0], alpha=0.5)
plt.tight_layout()
plt.show()
```



We discard the first half of the chain elements, thin the samples by a factor of 10, and flatten the resulted chain. We then proceed to plot the marginal posterior distributions:

[11]:
```
# flatten the chains, thin them by a factor of 10, and remove the burn-in (first half␣
↪of the chain)
chain = sampler.get_chain(flat=True, discard=nsteps//2, thin=10)

# plot marginal posterior distributions
fig, axes = zeus.cornerplot(chain, labels=['m', 'c'], truth=[m_true, c_true]);
```



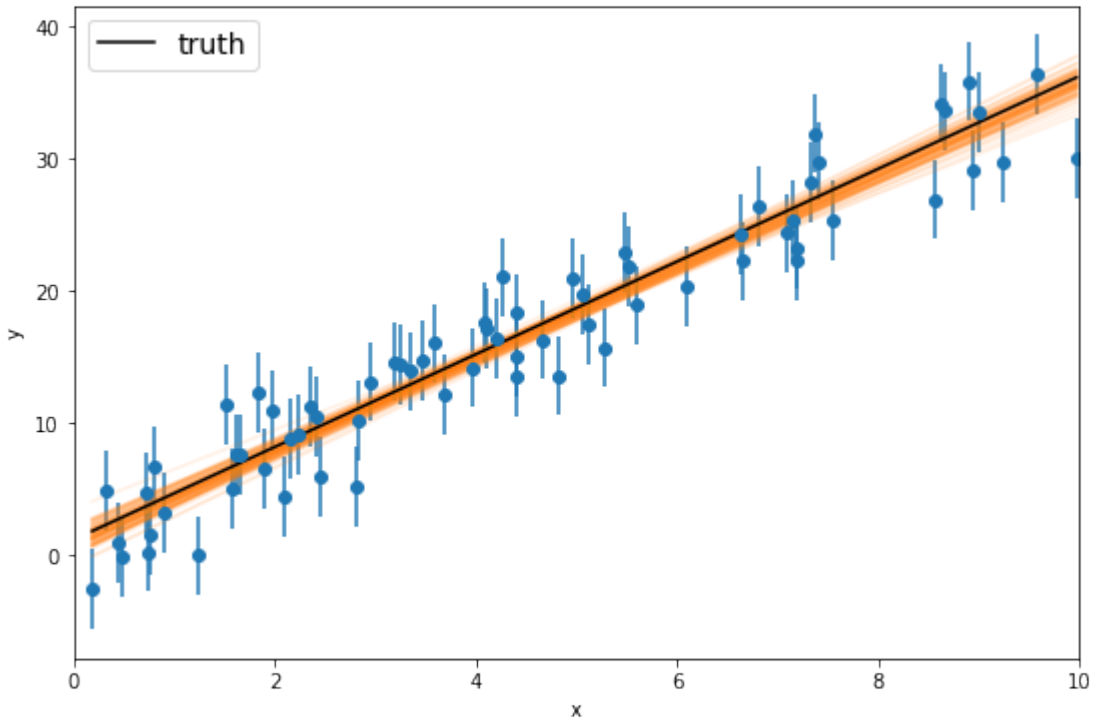
Now lets plot the projection of our results into the space of the observed data. The easiest way to do this is to randomly select 100 samples from the chain and plot the respective models on top the data points.

[12]:
```
inds = np.random.randint(len(chain), size=100)
plt.figure(figsize=(9,6))
for ind in inds:
    sample = chain[ind]
    plt.plot(x, np.dot(np.vander(x, 2), sample[:2]), "C1", alpha=0.1)
plt.errorbar(x, data, yerr=sigma, fmt="o")
plt.plot(x, straight_line(x,m_true,c_true), 'k', label="truth")
plt.legend(fontsize=14)
plt.xlim(0, 10)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("x")
plt.ylabel("y");
```



And finally we will print the *maximum a posteriori (MAP)* estimate along with the *1-sigma* uncertainty for the model parameters:

[13]:
```
labels=['m','c']
for i in range(ndim):
    mcmc = np.percentile(chain[:, 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))
```

$\mathrm{m} = 3.467^{0.122}_{-0.138}$

$\mathrm{c} = 1.032^{0.672}_{-0.643}$

Sampling from multimodal distributions

In this recipe we will demonstrate how one can use zeus with the Moves interface to sample efficiently from challenging high-dimensional multimodal distributions.

We will start by defining the target distribution, a 50-dimensional mixture of Normal distributions with huge valleys of almost-zero probability between the modes. This is an extremelly difficult target to sample from and most methods would fail.

[1]:
```
import zeus

import numpy as np
```

(continues on next page)

(continued from previous page)

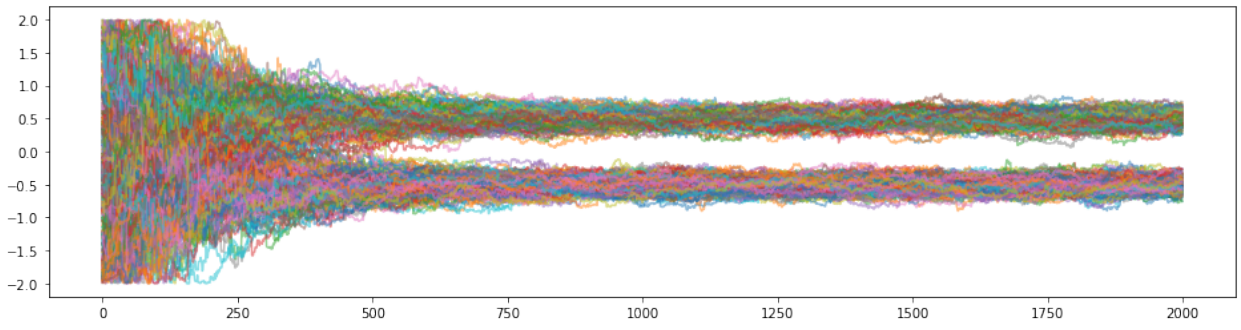
```
import matplotlib.pyplot as plt
import seaborn as sns

# Number of dimensions
ndim = 50

# Means
mu1 = np.ones(ndim) * (1.0 / 2)
mu2 = -mu1

# Standard Deviations
stdev = 0.1
sigma = np.power(stdev, 2) * np.eye(ndim)
isigma = np.linalg.inv(sigma)
dsigma = np.linalg.det(sigma)

w1 = 0.33  # one mode with 0.1 of the mass
w2 = 1 - w1 # the other mode with 0.9 of the mass

# Uniform prior limits
low = -2.0
high = 2.0

# The log-likelihood function of the Gaussian Mixture
def two_gaussians(x):
    log_like1 = (
        -0.5 * ndim * np.log(2 * np.pi)
        - 0.5 * np.log(dsigma)
        - 0.5 * (x - mu1).T.dot(isigma).dot(x - mu1)
    )
    log_like2 = (
        -0.5 * ndim * np.log(2 * np.pi)
        - 0.5 * np.log(dsigma)
        - 0.5 * (x - mu2).T.dot(isigma).dot(x - mu2)
    )
    return np.logaddexp.reduce([np.log(w1) + log_like1, np.log(w2) + log_like2])

# A simple uniform log-prior
def log_prior(x):
    if np.all(x>low) and np.all(x<high):
        return 0.0
    else:
        return -np.inf

# The Log-Posterior
def log_post(x):
    lp = log_prior(x)
    if not np.isfinite(lp):
        return -np.inf
    return lp + two_gaussians(x)
```

A failed attempt

Now lets run `zeus` for 1000 steps using 100 walkers and see what happens:

[2]:
```
nwalkers = 400
nsteps= 2000

# The starting positions of the walkers
start = low + np.random.rand(nwalkers,ndim) * (high - low)

# Initialise the Ensemble Sampler
sampler = zeus.EnsembleSampler(nwalkers, ndim, log_post)
# Run MCMC
sampler.run_mcmc(start, nsteps)

# Get the samples
samples = sampler.get_chain()

# Plot the walker trajectories for the first parameter of the 10
plt.figure(figsize=(16,4))
plt.plot(samples[:,:,0],alpha=0.5)
plt.show()
```

```
Initialising ensemble of 400 walkers...
Sampling progress : 100%|| 2000/2000 [03:05<00:00, 10.80it/s]
```



As you can see, once the walkers have found the modes/peaks of the Gaussian Mixture they stay stranded there, unable to jump to the other modes. This is a huge issue because it prevents the walkers from distributing the probability mass fairly among the peaks thus leading to biased results.

The clever way...

Now that we know that our target is multimodal, and that the default `DifferentialMove` cannot facilitate jumps between modes we can use a more advanced move such as the `GlobalMove`.

Although the `GlobalMove` is a very powerful tools, it is not well suited during the burnin phase. For that reason we will use the default `DifferentialMove` during burnin and then bring out the big guns.

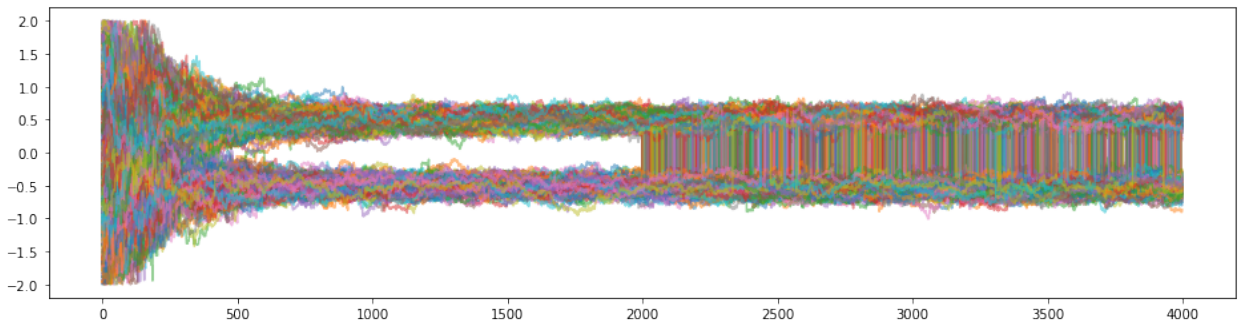
[3]:
```
# Initialise the Ensemble Sampler using the default ``DifferentialMove``
sampler = zeus.EnsembleSampler(nwalkers, ndim, log_post)
# Run MCMC
sampler.run_mcmc(start, nsteps)

# Get the burnin samples
burnin = sampler.get_chain()

# Set the new starting positions of walkers based on their last positions
start = burnin[-1]
```

(continues on next page)

(continued from previous page)

```
# Initialise the Ensemble Sampler using the advanced ``GlobalMove``.
sampler = zeus.EnsembleSampler(nwalkers, ndim, log_post, moves=zeus.moves.
↪GlobalMove())
# Run MCMC
sampler.run_mcmc(start, nsteps)

# Get the samples and combine them with the burnin phase for plotting purposes
samples = sampler.get_chain()
total_samples = np.concatenate((burnin, samples))

# Plot the walker trajectories for the first parameter of the 10
plt.figure(figsize=(16,4))
plt.plot(total_samples[:,:,0],alpha=0.5)
plt.show()
```

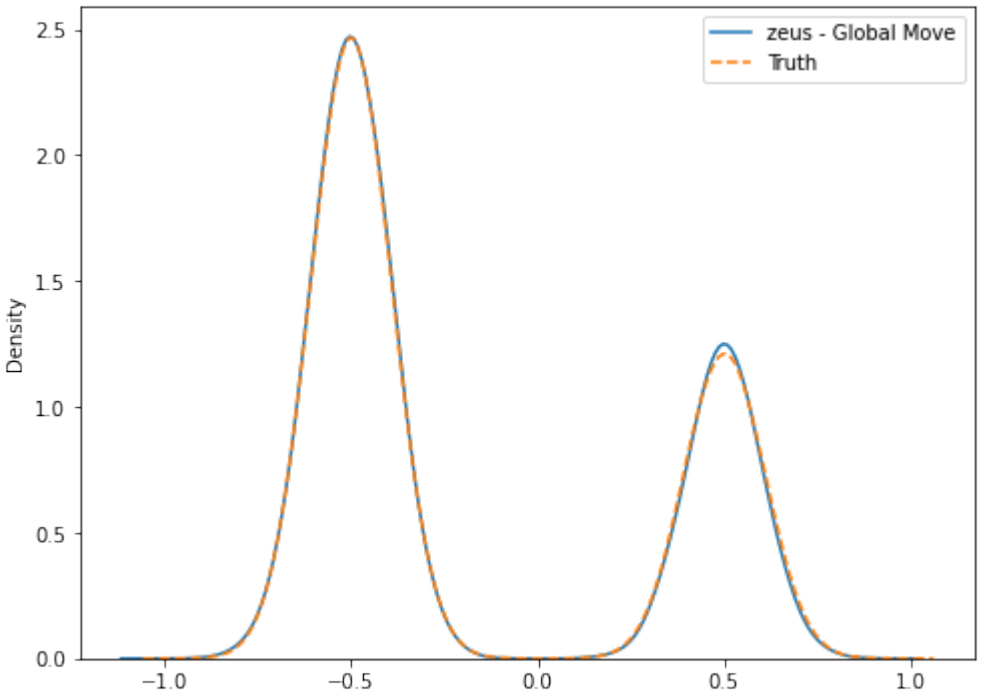
```
Initialising ensemble of 400 walkers...
Sampling progress : 100%|| 2000/2000 [03:03<00:00, 10.89it/s]
Initialising ensemble of 400 walkers...
Sampling progress : 100%|| 2000/2000 [06:52<00:00,  4.85it/s]
```



You can see that the moment we switched to the `GlobalMove` the walkers begun to jump from mode to mode frequently.

Lets now plot the 1D distribution of the first parameter and compare this with "actual truth".

[7]:
```
# Compute true samples from the gaussian mixture directly
s1 = np.random.multivariate_normal(mu1, sigma,size=int(w1*200000))
s2 = np.random.multivariate_normal(mu2, sigma,size=int(w2*200000))
samples_true = np.vstack((s1,s2))

# Get the chain from zeus
chain = sampler.get_chain(flat=True, discard=0.5)

# Plot Comparison
plt.figure(figsize=(8,6))
sns.kdeplot(chain[:,0])
sns.kdeplot(samples_true[:,0], ls='--')
plt.legend(['zeus - Global Move', 'Truth']);
```



Using the advanced moves, the walkers can move great distances in parameter space.

[ ]:

Parallelizing sampling using multiprocessing

We are going to use the multiprocessing Pool to parallelize and accelerate sampling.

This approach is ideal for personal computers, laptops, or small clusters and should work even in Jupyter notebooks.

In order to simulate a computationaly expensive log probability density function we will use the time package.

[1]:
```
import zeus
import numpy as np
import time
from multiprocessing import Pool
```

We define an uncorrelated normal distribution as our target distribution.

[2]:
```
ndim = 5
nwalkers = 2 * ndim
nsteps = 100

def log_prob(x):
    time.sleep(0.003)
    return -0.5 * np.sum(x**2.0)

start = np.random.randn(nwalkers, ndim)
```

We first run the sampler without in serial, without multiprocessing:

[3]:
```
t0 = time.time()

sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob)
sampler.run_mcmc(start, nsteps)

print("Serial took {0:.1f} seconds".format(time.time()-t0))
```

```
Initialising ensemble of 10 walkers...
Sampling progress : 100%|| 100/100 [00:19<00:00,  5.18it/s]
```

```
Serial took 19.3 seconds
```

And then run the sampler with multiprocessing:

[4]:
```
t0 = time.time()

with Pool() as pool:
    sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob, pool=pool)
    sampler.run_mcmc(start, nsteps)

print("Multiprocessing took {0:.1f} seconds".format(time.time()-t0))
```

```
Initialising ensemble of 10 walkers...
Sampling progress : 100%|| 100/100 [00:07<00:00, 12.93it/s]
```

```
Multiprocessing took 7.8 seconds
```

[ ]:
```

```

Parallelizing sampling using MPI

To take advantage of modern high performance computing facilities such as clusters with hundreds of CPUs we recommend to use `MPI` instead of `multiprocessing`.

To do this we will use the `ChainManager` included in `zeus`.

In order to run this example, copy and paste the following script into a file called 'test_mpi.py' and run the following command in the terminal:

```
mpiexec -n 8 python3 test_mpi.py
```

This will spawn 8 `MPI` processes and divide them into 2 independent chains of 10 walkers each. Unfortunately `MPI` is not compatible with `Jupyter` notebooks.

Save this as 'test_mpi.py'

```
import numpy as np
import zeus
from zeus import ChainManager

ndim = 5
nwalkers = 2 * ndim
nsteps = 100
```

(continues on next page)

(continued from previous page)

```
nchains = 2

def log_prob(x):
    return -0.5 * np.sum(x**2.0)

start = np.random.randn(nwalkers, ndim)


with ChainManager(nchains) as cm:
    rank = cm.get_rank

    sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob, pool=cm.get_pool)
    sampler.run_mcmc(start, nsteps)
    chain = sampler.get_chain(flat=True, discard=0.5)

    np.save('chain_'+str(rank)+'.npy', chain)
```

Blobs and Metadata

We introduce the blobs interface. An easy way for the user to track arbitrary metadata for every sample of the chain.

Tracking the value of the log-prior

We can easily use blobs to store the value of the log-prior at each step in the chain by doing something like:

[1]:
```
import zeus

import numpy as np

def log_prior(x):
    return -0.5 * np.dot(x,x)

def log_like(x):
    return -0.5 * np.dot(x,x) / 0.1**2.0

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

nwalkers, ndim = 32, 3
start = np.random.randn(nwalkers, ndim)
sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob)
sampler.run_mcmc(start, 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,)
```

```
Initialising ensemble of 32 walkers...
Sampling progress : 100%|| 100/100 [00:00<00:00, 160.45it/s](100, 32)
(3200,)
```

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

Tracking multiple species of metadata

When handling multiple species of metadata, it can be useful to name them. This can be done using the `blobs_dtype` argument of the `EnsembleSampler`. For instance, to save the mean of the parameters as well as the log-prior we could do something like:

[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, np.mean(params)

nwalkers, ndim = 32, 3
start = np.random.randn(nwalkers, ndim)

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

sampler.run_mcmc(start, 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)
```

```
Initialising ensemble of 32 walkers...
Sampling progress : 100%|| 100/100 [00:00<00:00, 137.06it/s](100, 32)
(100, 32)
(3200,)
(3200,)
```

[ ]:

Incrementally saving progress to a file

In many cases it is useful to save the chain to a file. This makes iit easier to post-process a long chain and makes things less disastrous if the computer crashes somewhere in the midle of an expensive MCMC run.

In this recipe we are going to use the callback interface to save the samples and their corresponding log-probability values in a `.h5` file. To do this you need to have `` `h5py <https://docs.h5py.org/en/latest/build.html#pre-built-installation-recommended>`__ `` installed.

We will set up a simple problem of sampling from a normal/Gaussian distribution as an example:

[1]:
```
import zeus
import numpy as np

ndim = 2
nwalkers = 10
nsteps = 1000

def log_prob(x):
    return -0.5*np.dot(x,x)

x0 = 1e-3 * np.random.randn(nwalkers, ndim)
```

Where `x0` is the initial positions of the walkers.

We will then initialise the sampler and start the MCMC run by providing the `zeus.callbacks.SaveProgressCallback` callback function.

[2]:
```
sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob)
sampler.run_mcmc(x0, nsteps, callbacks=zeus.callbacks.SaveProgressCallback("saved_
↪chains.h5", ncheck=100))
```

```
Initialising ensemble of 10 walkers...
Sampling progress : 100%|| 1000/1000 [00:01<00:00, 656.62it/s]
```

The above piece of code saved the chain incrementally every `ncheck=100` steps to a file named `saved_chains.h5`. We can now access the chains using the `h5py` package as follows:

[3]:
```
import h5py

with h5py.File('saved_chains.h5', "r") as hf:
    samples = np.copy(hf['samples'])
    logprob_samples = np.copy(hf['logprob'])

print(samples.shape)
print(logprob_samples.shape)
```

```
(1000, 10, 2)
(1000, 10)
```

Automated Convergence Diagnostics using the callback interface

Knowing when to stop sampling can be very useful when running expensive MCMC procedures. Ideally, if we want unbiased results, we want the sampler to stop after it has converged to the stationary phase (i.e. after the burn-in/warm-up period is over). To do this we can combine different `Convergence Diagnostics` offered as `callback functions` by zeus.

We will start by setting the simple problem of sampling from a bimodal Gaussian mixture distribution:

[105]:
```
import zeus
import numpy as np
import matplotlib.pyplot as plt

nsteps, nwalkers, ndim = 100000, 50, 5

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

x0 = 1e-3*np.random.randn(nwalkers,ndim) + 5.0
```

Where `nsteps` would be the maximum number of steps/iterations, `ivar` would be the inverse variance (precision) of the normal target distribution that we are going to sample from, and `x0` is the starting position of the walkers.

We will then define all the convergence diagnostics that we will use as `callback functions`.

First of all, we would like check the integrated autocorrelation time (IAT) of the chain every `ncheck=100` steps and make sure that we don't stop running unless the length of the chain is longer than `nact=50` times the IAT and that the rate of change of IAT drops bellow 1 percent (i.e. `dact=0.01`). We would also discard the first half of the chain (i.e. `discard=0.5`) before computing the IAT.

[106]:
```
cb0 = zeus.callbacks.AutocorrelationCallback(ncheck=100, dact=0.01, nact=50,␣
↪discard=0.5)
```

We will then use the **Split-R Gelman-Rubin statistic** computed using different segments (i.e. split into `nsplits=2` parts) of the same chain and decide that the sampler has converged if its value drops bellow `(1+epsilon)=1.01`.

[107]:
```
cb1 = zeus.callbacks.SplitRCallback(ncheck=100, epsilon=0.01, nsplits=2, discard=0.5)
```

Finally, just to make sure that the sampler doesn't stop too early, we will set the minimum number of iterations to `nmin=500`.

[108]:
```
cb2 = zeus.callbacks.MinIterCallback(nmin=500)
```

We are now ready to start sampling and require that all three of the aforementioned criteria are satisfied before sampling terminates.

[109]:
```
sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob)
sampler.run_mcmc(x0, nsteps, callbacks=[cb0, cb1, cb2])
```

```
Initialising ensemble of 50 walkers...
Sampling progress :   2%|         | 1898/100000 [00:13<13:42, 119.22it/s]
```

We noticed that the sampler automatically stopped running after approximately `1900` iterations. We can now have a look at the `split-R` statistics and the IAT estimate.

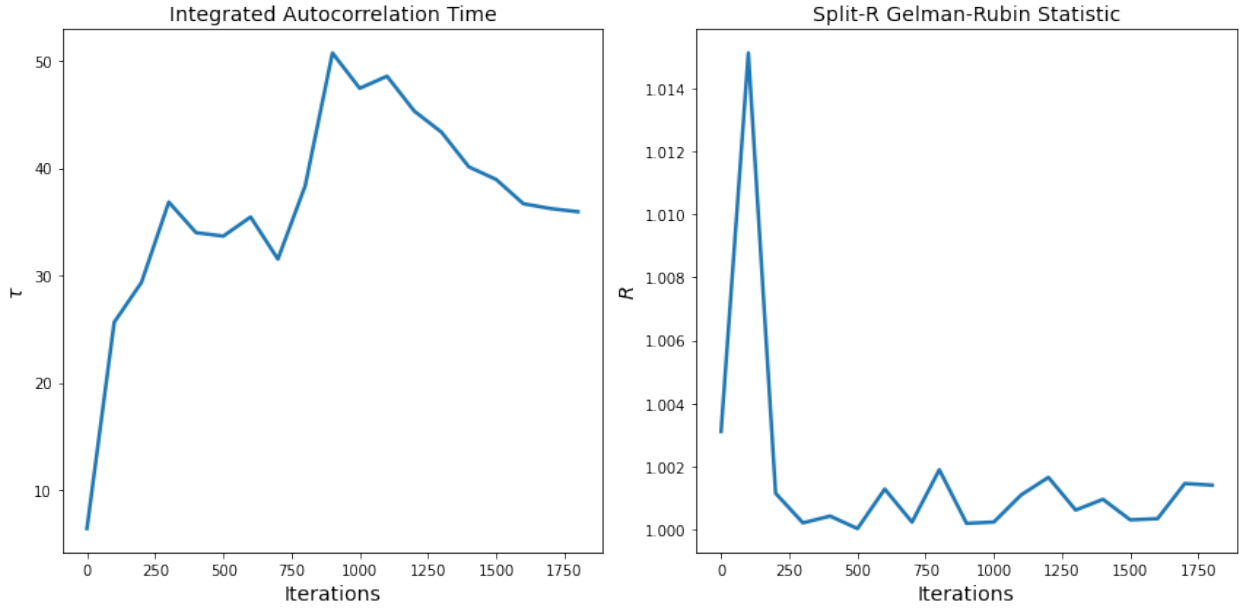
[110]:
```
tau = cb0.estimates
R = cb1.estimates

N = np.arange(len(tau)) * 100


plt.figure(figsize=(12,6))
plt.subplot(121)

plt.plot(N, tau, lw=2.5)
plt.title('Integrated Autocorrelation Time', fontsize=14)
plt.xlabel('Iterations', fontsize=14)
```

(continues on next page)

(continued from previous page)

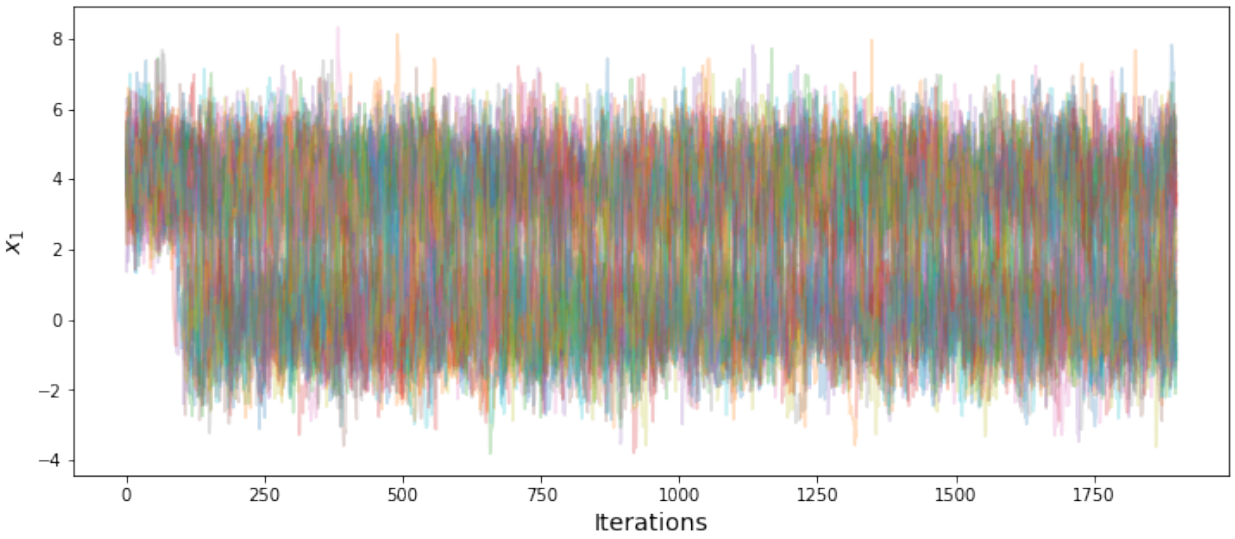
```
plt.ylabel(r'$\tau$', fontsize=14)


plt.subplot(122)

plt.plot(N, R, lw=2.5)
plt.title('Split-R Gelman-Rubin Statistic', fontsize=14)
plt.xlabel('Iterations', fontsize=14)
plt.ylabel(r'$R$', fontsize=14)

plt.tight_layout()
plt.show()
```

```
Sampling progress :   2%|         | 1900/100000 [00:13<11:50, 138.00it/s]
```



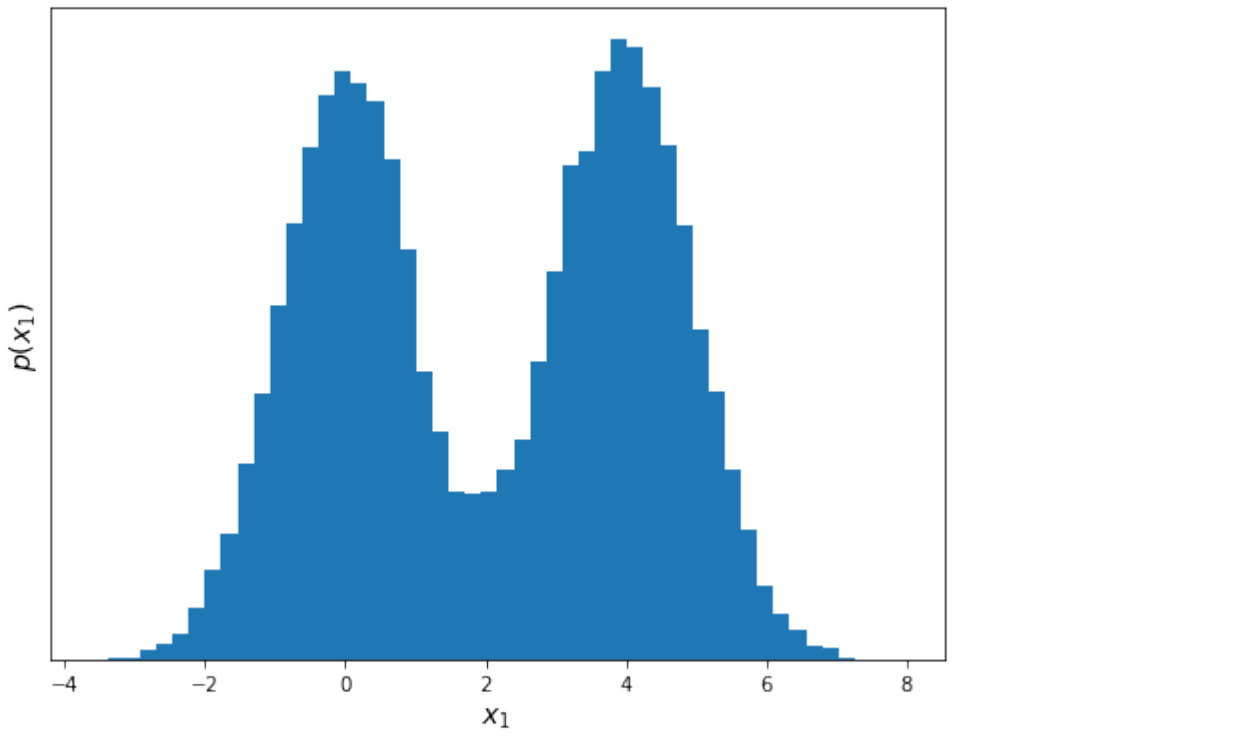
We can also have a look at the traces of the walkers.

[111]:
```
samples = sampler.get_chain()

plt.figure(figsize=(12,5))
plt.plot(samples[:,:,0],alpha=0.25)
plt.xlabel('Iterations', fontsize=14)
plt.ylabel(r'$x_{1}$', fontsize=14)
plt.show()
```



And also the 1-dimensional marginal distribution of the first parameter.

[112]:
```
chain = sampler.get_chain(flat=True, discard=0.5)

plt.figure(figsize=(8,6))
plt.hist(chain[:,0], 50)
plt.gca().set_yticks([])
plt.xlabel(r"$x_{1}$", fontsize=14)
plt.ylabel(r"$p(x_{1})$", fontsize=14)
plt.show()
```



Parallel sampling using MPI and Gelman-Rubin convergence diagnostics

To take advantage of modern high performance computing facilities such as clusters with hundreds of CPUs we recommend to use `MPI` instead of `multiprocessing`.

To do this we will use the `ChainManager` included in `zeus`. We will also use the `ParallelSplitRCallback` function to check the Gelman-Rubing convergence diagnostic during the run and terminate sampling automatically.

In order to run this example, copy and paste the following script into a file called 'test_mpi.py' and run the following command in the terminal:

```
mpiexec -n 8 python3 test_mpi_gr.py
```

This will spawn 8 `MPI` processes and divide them into 2 independent chains of 10 walkers each. Unfortunately `MPI` is not compatible with `Jupyter` notebooks.

Save this as 'test_mpi_gr.py'

```
import numpy as np
import zeus
from zeus import ChainManager

ndim = 20
nwalkers = 2 * ndim
nsteps = 10000
nchains = 2

def log_prob(x):
    return -0.5 * np.sum(x**2.0)

start = 1e-2 * np.random.randn(nwalkers, ndim)  + 20.0


with ChainManager(nchains) as cm:
    rank = cm.get_rank

    cb = zeus.callbacks.ParallelSplitRCallback(epsilon=0.01, chainmanager=cm)
    sampler = zeus.EnsembleSampler(nwalkers, ndim, log_prob, pool=cm.get_pool)
    sampler.run_mcmc(start, nsteps, callbacks=cb)
    chain = sampler.get_chain(flat=True, discard=0.5)

    if rank == 0:
        print('R =', cb.estimates, flush=True)
    np.save('chain_'+str(rank)+'.npy', chain)
```

5.2 Frequently Asked Questions

5.2.1 What is the acceptance rate of `zeus`?

Unlike most MCMC methods, `zeus` acceptance rate isn't varying during a run. As a matter of fact, its acceptance rate is identically 1, always. This is because of the Slice Sampler at its core.

5.2.2 Why should I use zeus instead of other MCMC samplers?

The first reason you should think of using `zeus` is due to the fact that it doesn't require any hand tuning at all. There is no need to adjust any hyperparameters or provide a proposal distribution.

Moreover, unlike other black-box MCMC methods `zeus` is more robust to the curse of dimensionality and handle challenging distributions better.

5.2.3 What are the walkers?

Walkers are the members of the ensemble. They are interacting parallel chains which collectively explore the posterior mass.

5.2.4 How many walkers should I use?

At least twice the number of parameters of your problem. A good rule of thump is to use between 2 and 4 times the number of parameters. If your distribution has multiple modes/peaks you may want to increase the number of walkers.

5.2.5 How should I initialize the positions of the walkers?

A good practice seems to be to initialize the walkers from a small ball close to the *Maximum a Posteriori* estimate. After a few autocorrelation times the walkers would have explored the rest of the usefull regions of the parameter space (i.e. the typical set), producing a great number of independent samples.

5.2.6 How long should I run `zeus`?

You don't have to run `zeus` for very long. If your goal is to produce 2D/1D contours and/or 1-sigma/2-sigma constraints for your parameters, running `zeus` for a few autocorrelation times (e.g. 10) is more than enough. You can also use the implemented callback functions (see Cookbook and API) to automate the termination of a run.

5.2.7 What can I do if the first few iterations take too long to complete?

This usually occurs when the walkers are initialised closed to each other. During the first `10-100` iterations `zeus` is tuning its proposal scale `mu`. During that time `zeus` may do more model evaluations than usual. Tuning of `mu` is faster if initialised from a large value. We thus recommend to set `mu` to an large value (e.g. `mu=1e3`) initially in the `EnsembleSampler`.

5.2.8 Is there any way to reduce the computational cost per iteration?

`zeus`'s power originates in its flexibility. During each iteration, the walkers move along straight lines (i.e. slices) that cross the posterior mass. The construction of a slice involves two steps, an initial expanding/stepping-out and a subsequent shrinking procedure. One can decrease the computational cost per iteration by forcing `zeus` to conduct no expansions. This is achieved by setting `light_mode=True` in the `EnsembleSampler` at the cost of reduced flexibility. If the target distribution is close to normal/Gaussian one then this procedure can cut the cost to half.

5.2.9 What are the `Moves` and which one should I use?

`zeus` was originally built on the `Differential` and `Gaussian` moves. Starting from version 2.0.0, `zeus` supports a mixture of different moves/proposals. Moves are recipes that the walkers follow to cross the parameter space. The `Differential Move` remains the default choice but we also provide a suite of additional moves, such as the `Global Move` that can be used when sampling from challenging target distributions (e.g. highly dimensional multimodal distributions).

The move(s) you should use depends on the particular target distribution. The `Differential Move` seems to be a good choice for most distributions and 50-50 mixture of the `Global Move` and `Local Move` seem to perform very well in highly dimensional multimodal distributions when used after the burnin period is over.

5.3 API Reference

zeus consists mainly of six parts:

5.3.1 The Ensemble Slice Sampler

class zeus.**EnsembleSampler**(*nwalkers*, *ndim*, *logprob_fn*, *args=None*, *kwargs=None*, *moves=None*, *tune=True*, *tolerance=0.05*, *patience=5*, *maxsteps=10000*, *mu=1.0*, *maxiter=10000*, *pool=None*, *vectorize=False*, *blobs_dtype=None*, *verbose=True*, *check_walkers=True*, *shuffle_ensemble=True*, *light_mode=False*)

An Ensemble Slice Sampler.

Parameters

- **nwalkers** (*int*) – The number of walkers in the ensemble.
- **ndim** (*int*) – The number of dimensions/parameters.
- **logprob_fn** (*callable*) – A python function that takes a vector in the parameter space as input and returns the natural logarithm of the unnormalised posterior probability at that position.
- **args** (*list*) – Extra arguments to be passed into the logp.
- **kwargs** (*list*) – Extra arguments to be passed into the logp.
- **moves** (*list*) – This can be a single move object, a list of moves, or a "weighted" list of the form `[(zeus.moves.DifferentialMove(), 0.1), ...]`. When running, the sampler will randomly select a move from this list (optionally with weights) for each proposal. (default: DifferentialMove)
- **tune** (*bool*) – Tune the scale factor to optimize performance (Default is True.)
- **tolerance** (*float*) – Tuning optimization tolerance (Default is 0.05).
- **patience** (*int*) – Number of tuning steps to wait to make sure that tuning is done (Default is 5).
- **maxsteps** (*int*) – Number of maximum stepping-out steps (Default is 10^4).
- **mu** (*float*) – Scale factor (Default value is 1.0), this will be tuned if tune=True.
- **maxiter** (*int*) – Number of maximum Expansions/Contractions (Default is 10^4).
- **pool** (*bool*) – External pool of workers to distribute workload to multiple CPUs (default is None).

- **vectorize** (*bool*) – If true (default is False), logprob_fn receives not just one point but an array of points, and returns an array of log-probabilities.
- **blobs_dtype** (*list*) – List containing names and dtypes of blobs metadata e.g. `[("log_prior", float), ("mean", float)]`. It's useful when you want to save multiple species of metadata. Default is None.
- **verbose** (*bool*) – If True (default) print log statements.
- **check_walkers** (*bool*) – If True (default) then check that `nwalkers >= 2*ndim` and even.
- **shuffle_ensemble** (*bool*) – If True (default) then shuffle the ensemble of walkers in every iteration before splitting it.
- **light_mode** (*bool*) – If True (default is False) then no expansions are performed after the tuning phase. This can significantly reduce the number of log likelihood evaluations but works best in target distributions that are apprroximately Gaussian.

act
Integrated Autocorrelation Time (IAT) of the Markov Chain.

Returns Array with the IAT of each parameter.

chain
Returns the chains.

Returns Returns the chains of shape (nsteps, nwalkers, ndim).

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.

Returns 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.

Return type log_prob

efficiency
Effective Samples per Log Probability Evaluation.

Returns efficiency

ess
Effective Sampling Size (ESS) of the Markov Chain.

Returns ESS

get_blobs (*flat=False*, *thin=1*, *discard=0*)
Get the values of the blobs at each step of the chain.

Parameters

- **flat** (*bool*) – If True then flatten the chain into a 1D array by combining all walkers (default is False).
- **thin** (*int*) – Thinning parameter (the default value is 1).
- **discard** (*int*) – Number of burn-in steps to be removed from each walker (default is 0). A float number between 0.0 and 1.0 can be used to indicate what percentage of the chain to be discarded as burnin.

Returns (structured) numpy array containing the values of the blobs at each step of the chain.

get_chain(*flat=False*, *thin=1*, *discard=0*)
Get the Markov chain containing the samples.

Parameters

- **flat** (*bool*) – If True then flatten the chain into a 2D array by combining all walkers (default is False).
- **thin** (*int*) – Thinning parameter (the default value is 1).
- **discard** (*int*) – Number of burn-in steps to be removed from each walker (default is 0). A float number between 0.0 and 1.0 can be used to indicate what percentage of the chain to be discarded as burnin.

Returns Array object containg the Markov chain samples (2D if flat=True, 3D if flat=False).

get_last_blobs()
Return the blobs for the last position of the walkers.

get_last_log_prob()
Return the log probability values for the last position of the walkers.

get_last_sample()
Return the last position of the walkers.

get_log_prob(*flat=False*, *thin=1*, *discard=0*)
Get the value of the log probability function evalutated at the samples of the Markov chain.

Parameters

- **flat** (*bool*) – If True then flatten the chain into a 1D array by combining all walkers (default is False).
- **thin** (*int*) – Thinning parameter (the default value is 1).
- **discard** (*int*) – Number of burn-in steps to be removed from each walker (default is 0). A float number between 0.0 and 1.0 can be used to indicate what percentage of the chain to be discarded as burnin.

Returns Array containing the value of the log probability at the samples of the Markov chain (1D if flat=True, 2D otherwise).

ncall
Number of Log Prob calls.

Returns ncall

reset()
Reset the state of the sampler. Delete any samples stored in memory.

run_mcmc(*start*, *nsteps=1000*, *thin=1*, *progress=True*, *log_prob0=None*, *blobs0=None*, *thin_by=1*, *callbacks=None*)
Run MCMC.

Parameters

- **start** (*float*) – Starting point for the walkers. If None then the sampler proceeds from the last known position of the walkers.
- **nsteps** (*int*) – Number of steps/generations (default is 1000).
- **thin** (*float*) – Thin the chain by this number (default is 1, no thinning).
- **progress** (*bool*) – If True (default), show progress bar.

- **log_prob0** (*float*) – Log probability values of the walkers. Default is `None`.
- **blobs0** (*float*) – Blob value of the walkers. Default is `None`.
- **thin_by** (*float*) – 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.
- **callbacks** (*function*) – Callback function or list with multiple callback actions (e.g. `[callback_0, callback_1, ...]`) to be evaluated during the run. Sampling terminates when all of the callback functions return `True`. This option is useful in cases in which sampling needs to terminate once convergence is reached. Examples of callback functions can be found in the API docs.

sample (*start*, *log_prob0=None*, *blobs0=None*, *iterations=1*, *thin=1*, *thin_by=1*, *progress=True*)

Advance the chain as a generator. The current iteration index of the generator is given by the `sampler.iteration` property.

Parameters

- **start** (*float*) – Starting point for the walkers.
- **log_prob0** (*float*) – Log probability values of the walkers. Default is `None`.
- **blobs0** (*float*) – Blob value of the walkers. Default is `None`.
- **iterations** (*int*) – Number of steps to generate (default is 1).
- **thin** (*float*) – Thin the chain by this number (default is 1, no thinning).
- **thin_by** (*float*) – 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.
- **progress** (*bool*) – If True (default), show progress bar.

scale_factor

Scale factor values during tuning.

Returns scale factor mu

summary

Summary of the MCMC run.

5.3.2 The Callbacks

Starting from version 2.4.0, `zeus` supports callback functions. Those are functions that are called in every iteration of a run. Among other things, these can be used to monitor useful quantities, assess convergence, and save the chains to disk. Custom callback functions can also be used. Sampling terminates if a callback function returns `True` and continues running while `False` or `None` is returned.

Autocorrelation Callback

class zeus.callbacks.**AutocorrelationCallback**(*ncheck=100*, *dact=0.01*, *nact=10*, *discard=0.5*, *trigger=True*, *method='mk'*)

The Autocorrelation Time Callback class checks the integrated autocorrelation time (IAT) of the chain during the run and terminates sampling if the rate of change of IAT is below some threshold and the length of the chain is greater than some multiple of the IAT estimate.

Parameters

- **ncheck** (*int*) – The number of steps after which the IAT is estimated and the tests are performed. Default is `ncheck=100`.
- **dact** (*float*) – Threshold of the rate of change of IAT. Sampling terminates once this threshold is reached along with the other criteria. Default is `dact=0.01`.
- **nact** (*float*) – Minimum lenght of the chain as a mutiple of the IAT. Sampling terminates once this threshold is reached along with the other criteria. Default is `nact=10`.
- **discard** (*float*) – Percentage of chain to discard prior to estimating the IAT. Default is `discard=0.5`.
- **trigger** (*bool*) – If `True` (default) then terminatate sampling once converged, else just monitor statistics.
- **method** (*str*) – Method to use for the estimation of the IAT. Available options are `mk` (Default), `dfm`, and `gw`.

Split-R Callback

class zeus.callbacks.**SplitRCallback**(*ncheck=100*, *epsilon=0.05*, *nsplits=2*, *discard=0.5*, *trigger=True*)

The Split-R Callback class checks the Gelman-Rubin criterion during the run by splitting the chain into multiple parts and terminates sampling if the Split-R coefficient is close to unity.

Parameters

- **ncheck** (*int*) – The number of steps after which the Gelman-Rubin statistics is estimated and the tests are performed. Default is `ncheck=100`.
- **epsilon** (*float*) – Threshold of the Split-R value. Sampling terminates when `|R-1|<epsilon`. Default is `0.05`
- **nsplits** (*int*) – Split each chain into this many pieces. Default is `2`.
- **discard** (*float*) – Percentage of chain to discard prior to estimating the IAT. Default is `discard=0.5`.
- **trigger** (*bool*) – If `True` (default) then terminatate sampling once converged, else just monitor statistics.

Parallel Split-R Callback

class zeus.callbacks.**ParallelSplitRCallback**(*ncheck=100*, *epsilon=0.01*, *nsplits=2*, *discard=0.5*, *trigger=True*, *chainmanager=None*)

The Parallel Split-R Callback class extends the functionality of the Split-R Callback to more than one CPUs by checking the Gelman-Rubin criterion during the run by splitting the chain into multiple parts and combining different parts from parallel chains and terminates sampling if the Split-R coefficient is close to unity.

Parameters

- **ncheck** (*int*) – The number of steps after which the Gelman-Rubin statistics is estimated and the tests are performed. Default is `ncheck=100`.
- **epsilon** (*float*) – Threshold of the Split-R value. Sampling terminates when `|R-1|<epsilon`. Default is `0.05`
- **nsplits** (*int*) – Split each chain into this many pieces. Default is `2`.

- **discard** (*float*) – Percentage of chain to discard prior to estimating the IAT. Default is `discard=0.5`.
- **trigger** (*bool*) – If `True` (default) then terminatate sampling once converged, else just monitor statistics.
- **chainmanager** (*ChainManager instance*) – The `ChainManager` used to parallelise the sampling process.

Minimum Iterations Callback

class zeus.callbacks.**MinIterCallback**(*nmin=1000*)

The Minimum Iteration Callback class ensure that sampling does not terminate early prior to a prespecified number of steps.

Parameters **nmin** (*int*) – The number of minimum steps before other callbacks can terminate the run.

Save Progress Callback

class zeus.callbacks.**SaveProgressCallback**(*filename='./chains.h5'*, *ncheck=100*)

The Save Progress Callback class iteratively saves the collected samples and log-probability values to a HDF5 file.

Parameters

- **filename** (*str*) – Name of the directory and file to save samples. Default is `./chains.h5`.
- **ncheck** (*int*) – The number of steps after which the samples are saved. Default is `ncheck=100`.

5.3.3 The Ensemble Moves

`zeus` was originally built on the `Differential` and `Gaussian` moves. Starting from version 2.0.0, `zeus` supports a mixture of different moves/proposals. Moves are recipes that the walkers follow to cross the parameter space. The `Differential Move` remains the default choice but we also provide a suite of additional moves, such as the `Global Move` that can be used when sampling from challenging target distributions (e.g. highly dimensional multimodal distributions).

Differential Move

class zeus.moves.**DifferentialMove**(*tune=True*, *mu0=1.0*)

The Karamanis & Beutler (2020) "Differential Move" with parallelization. When this Move is used the walkers move along directions defined by random pairs of walkers sampled (with no replacement) from the complementary ensemble. This is the default choice and performs well along a wide range of target distributions.

Parameters

- **tune** (*bool*) – If True then tune this move. Default is True.
- **mu0** (*float*) – Default value of `mu` if `tune=False`.

get_direction(*X*, *mu*)

Generate direction vectors.

Parameters

- **X** (*array*) – Array of shape `(nwalkers//2, ndim)` with the walker positions of the complementary ensemble.
- **mu** (*float*) – The value of the scale factor `mu`.

Returns **directions** – Array of direction vectors of shape `(nwalkers//2, ndim)`.

Return type array

Gaussian Move

class zeus.moves.**GaussianMove**(*tune=False*, *mu0=1.0*, *cov=None*)

The Karamanis & Beutler (2020) "Gaussian Move" with parallelization. When this Move is used the walkers move along directions defined by random vectors sampled from the Gaussian approximation of the walkers of the complementary ensemble.

Parameters

- **tune** (*bool*) – If True then tune this move. Default is True.
- **mu0** (*float*) – Default value of `mu` if `tune=False`.

get_direction(*X*, *mu*)

Generate direction vectors.

Parameters

- **X** (*array*) – Array of shape `(nwalkers//2, ndim)` with the walker positions of the complementary ensemble.
- **mu** (*float*) – The value of the scale factor `mu`.

Returns **directions** – Array of direction vectors of shape `(nwalkers//2, ndim)`.

Return type array

Global Move

class zeus.moves.**GlobalMove**(*tune=True*, *mu0=1.0*, *rescale_cov=0.001*, *n_components=5*)

The Karamanis & Beutler (2020) "Global Move" with parallelization. When this Move is used a Bayesian Gaussian Mixture (BGM) is fitted to the walkers of complementary ensemble. The walkers move along random directions which connect different components of the BGM in an attempt to facilitate mode jumping. This Move should be used when the target distribution is multimodal. This move should be used after any burnin period.

Parameters

- **tune** (*bool*) – If True then tune this move. Default is True.
- **mu0** (*float*) – Default value of `mu` if `tune=False`.
- **rescale_cov** (*float*) – Rescale the covariance matrices of the BGM components by this factor. This promotes mode jumping. Default value is 0.001.
- **n_components** (*int*) – The number of mixture components. Depending on the distribution of the walkers the model can decide not to use all of them.

get_direction(*X*, *mu*)

Generate direction vectors.

Parameters

- **X** (*array*) – Array of shape `(nwalkers//2, ndim)` with the walker positions of the complementary ensemble.
- **mu** (*float*) – The value of the scale factor `mu`.

Returns **directions** – Array of direction vectors of shape `(nwalkers//2, ndim)`.

Return type array

KDE Move

class `zeus.moves.`**KDEMove**(*tune=False*, *mu0=1.0*, *bw_method=None*)

The Karamanis & Beutler (2020) "KDE Move" with parallelization. When this Move is used the distribution of the walkers of the complementary ensemble is traced using a Gaussian Kernel Density Estimation methods. The walkers then move along random direction vectos sampled from this distribution.

Parameters

- **tune** (*bool*) – If True then tune this move. Default is True.
- **mu0** (*float*) – Default value of `mu` if `tune=False`.
- **bw_method** – The bandwidth estimation method. See the scipy docs for allowed values.

get_direction(*X*, *mu*)

Generate direction vectors.

Parameters

- **X** (*array*) – Array of shape `(nwalkers//2, ndim)` with the walker positions of the complementary ensemble.
- **mu** (*float*) – The value of the scale factor `mu`.

Returns **directions** – Array of direction vectors of shape `(nwalkers//2, ndim)`.

Return type array

Random Move

class `zeus.moves.`**RandomMove**(*tune=True*, *mu0=1.0*)

The Karamanis & Beutler (2020) "Random Move" with parallelization. When this move is used the walkers move along random directions. There is no communication between the walkers and this Move corresponds to the vanilla Slice Sampling method. This Move should be used for debugging purposes only.

Parameters

- **tune** (*bool*) – If True then tune this move. Default is True.
- **mu0** (*float*) – Default value of `mu` if `tune=False`.

get_direction(*X*, *mu*)

Generate direction vectors.

Parameters

- **X** (*array*) – Array of shape `(nwalkers//2, ndim)` with the walker positions of the complementary ensemble.
- **mu** (*float*) – The value of the scale factor `mu`.

Returns **directions** – Array of direction vectors of shape `(nwalkers//2, ndim)`.

Return type array

5.3.4 Autocorrelation Time Estimation

zeus.**AutoCorrTime**(*samples*, *c=5.0*, *method='mk'*)
Integrated Autocorrelation Time (IAT) for all the chains.

Parameters

- **samples** (*array*) – 3-dimensional array of shape (nsteps, nwalkers, ndim)
- **c** (*float*) – Truncation parameter of automated windowing procedure of Sokal (1989), default is 5.0
- **method** (*str*) – Method to use to compute the IAT. Available options are `mk` (Default), `dfm`, and `gw`.

Returns **taus** – Array with the IAT of all the chains.

Return type array

5.3.5 The Chain Manager & MPI Tools

The `Chain Manager` can be used to parallelize `zeus`. The benefits of this appoach is that the `Chain Manager` can parallelize many chains and walkers simultaneously. See the Cookbook for more information.

class zeus.**ChainManager**(*nchains=1*, *comm=None*)
Class to serve as context manager to handle to MPI-related issues, specifically, the managing of `MPIPool` and splitting of communicators. This class can be used to run `nchains` in parallel with each chain having its own `MPIPool` of parallel walkers.

Parameters

- **nchains** (*int*) – the number of independent chains to run concurrently
- **comm** (*MPI.Communicator*) – the global communicator to split

allgather(*x*)
Allgather method to gather `x` in all chains. This is equivalent to first `scatter` and then `bcast`.

Parameters **x** (*Python object*) – The python object to be gathered.

Returns **x** – The python object, gathered in all ranks.

Return type Python object

bcast(*x*, *root*)
Broadcast method to send `x` from `rank = root` to all chains.

Parameters

- **x** (*Python object*) – The python object to be send.
- **root** (*int*) – The rank of the origin chain from which the object x is sent.

Returns **x** – The input object x in all ranks.

Return type Python object

gather(*x*, *root*)
Gather method to gather `x` in `rank = root` chain.

Parameters

- **x** (*Python object*) – The python object to be gathered.
- **root** (*int*) – The rank of the chain that x is gathered.

Returns **x** – The input object x gathered in `rank = root`.

Return type Python object

get_pool
Get parallel `pool` of workers that correspond to a specific chain. This should be used to parallelize the walkers of each `chain` (not the chains themselves). This includes the `map` method that `zeus` requires.

get_rank
Get `rank` of current `chain`. The minimum `rank` is `0` and the maximum is `nchains-1`.

scatter(*x*, *root*)
Scatter method to scatter `x` from `rank = root` chain to the rest.

Parameters

- **x** (*Python object*) – The python object to be scattered.
- **root** (*int*) – The rank of the origin chain from which the x is scattered.

Returns **x** – Part of the input object x that was scattered along the ranks.

Return type Pythonn object

5.3.6 Plotting Results

Cornerplot

zeus.**cornerplot**(*samples, labels=None, weights=None, levels=None, span=None, quantiles=[0.025, 0.5, 0.975], truth=None, color=None, alpha=0.5, linewidth=1.5, fill=True, fontsize=10, show_titles=True, title_fmt='.2f', title_fontsize=12, cut=3, fig=None, size=(10, 10)*)
Plot corner-plot of samples.

Parameters

- **samples** (*array*) – Array of shape (nsamples, ndim) containing the samples.
- **labels** (*list*) – List of names of for the parameters.
- **weights** (*array*) – Array with weights (useful if different samples have different weights e.g. as in Nested Sampling).
- **levels** (*list*) – The quantiles used for plotting the smoothed 2-D distributions. If not provided, these default to 0.5, 1, 1.5, and 2-sigma contours.
- **quantiles** (*list*) – A list of fractional quantiles to overplot on the 1-D marginalized posteriors as titles. Default is `[0.025, 0.5, 0.975]` (spanning the 95%/2-sigma credible interval).
- **truth** (*array*) – Array specifying a point to be highlighted in the plot. It can be the true values of the parameters, the mean, median etc. By default this is None.
- **color** (*str*) – Matplotlib color to be used in the plot.
- **alpha** (*float*) – Transparency value of figure (Default is 0.5).
- **linewidth** (*float*) – Linewidth of plot (Default is 1.5).
- **fill** (*bool*) – If True (Default) the fill the 1D and 2D contours with color.
- **fontsize** (*float*) – Fontsize of axes labels. Default is 10.

- **show_titles** (*bool*) – Whether to display a title above each 1-D marginalized posterior showing the quantiles. Default is True.
- **title_fmt** (*str*) – Format of the titles. Default is `.2f`.
- **title_fontsize** (*float*) – Fontsize of titles. Default is 12.
- **cut** (*float*) – Factor, multiplied by the smoothing bandwidth, that determines how far the evaluation grid extends past the extreme datapoints. When set to 0, truncate the curve at the data limits. Default is `cut=3`.
- **fig** (*(figure, axes)*) – Pre-existing Figure and Axes for the plot. Otherwise create new internally. Default is None.
- **size** (*(int, int)*) – Size of the plot. Default is (10, 10).

Returns The matplotlib figure and axes.

Return type Figure, Axes

Index