Appendix A

Markov Chain Monte Carlo

Markov chain Monte Carlo (MCMC) algorithms aim to draw random samples from a target probability distribution (usually the posterior distribution, in a Bayesian analysis). This is achieved by constructing a *Markov chain* which, after many iterations, converges to the target distribution.

A *Markov chain* is a random sequence of points where the probability of each point depends only on the previous point in the chain.

Various algorithms exist for constructing such a Markov chain. The simplest and most well-known is the *Metropolis–Hastings* algorithm which is described here. The Metropolis–Hastings algorithm is very flexible and, as we shall see, is extremely simple to implement. It can be used to sample from any distribution $P(x)$; in particular, we do not care about the dimensionality (x can be vector of any length) and we do *not* require that $P(x)$ be properly normalised. (This later point is useful because, as we have seen, normalising posterior distributions in high dimensions involves evaluating the tricky evidence integral.)

The algorithm generates a sequence points, called a *chain*, $x_1, x_2, \ldots x_n$. The construction of this chain is designed such that, in the limit $n \to \infty$, the distribution of these points matches the target distribution $P(x)$; i.e. $x_i \sim P(x)$. The points are produced iteratively, and the location of the next point x_{i+1} depends only on the current point x_i (i.e. a Markov chain). At each iteration x_i, the algorithm randomly proposes a possible next point u. This proposed point is either accepted (in which case the chain moves to the prosed point; $x_{i+1} = u$) or rejected (in which case the chain stays where it is; $x_{i+1} = x_n$). The probability of the proposed point being accepted is decided by the relative values of the probability

density at the two points: $P(u)$ and $P(x_i)$. If $P(u) > P(x_i)$ then the chain will always move to u, if $P(u) < P(x_i)$ then the chain will only move to u with probability $P(u)/P(x_i)$; i.e. the chain "wants" to move uphill, but can sometimes move downhill too.

The algorithm requires us to specify a proposal distribution to choose the point u. This distribution is denoted Q, and at each step the proposed point u is drawn randomly from this distribution, $u \sim Q(u|x)$. Here, we will assume that this proposal is symmetric, i.e. $Q(x|u) = Q(u|x)$.

The algorithm then proceeds as follows.

1. **Initialise.** Pick an arbitrary starting value, x_0.

2. **Iterate.** For $i = 0, 1, 2, \ldots$. Generate a proposed point $u \sim Q(u|x_i)$, evaluate the *acceptance ratio* $\alpha = P(u)/P(x_i)$, and accept the proposed point (i.e. $x_{i+1} = u$) with probability $\min(\alpha, 1)$ otherwise reject the proposed point (i.e. $x_{i+1} = x_i$).

The result of the algorithm is a Markov chain of points $\{x_0, x_1, x_2 \ldots\}$ which should be distributed according to the target distribution $P(x)$; i.e. $x_i \sim P(x)$. Although, see the practical discussion of thinning and burnin below.

A.1 An example MCMC implementation

Let's illustrate the Metropolis–Hastings algorithm by implementing it ourselves for a simple example distribution in 2-dimensions; $\vec{x} = (x, y)$. Let us suppose we wish to sample points from the following target distribution which looks like a ring of radius $r_0 = 5$ in the 2D plane.

```
import numpy as np

def P(x, r0=5, w=1):
    """
    Target distribution, a ring in the 2D plane
    """
    r = np.linalg.norm(x)
    return np.exp(-0.5*((r-r0)/w)**2)
```

Notice that the above probability distribution $P(x)$ is *not* properly normalised; this does not matter for the Metropolis–Hastings algorithm.

We choose to use a Gaussian proposal distribution $Q(u|x)$; i.e.$u = \mathcal{N}(x, I_2)$, where I_2 is the 2×2 identity matrix.

```
from scipy.stats import multivariate_normal

def Q(x, cov=np.identity(2)):
    """
    proposal distribution, a 2D Gaussian centred on the current point
    """
    u = multivariate_normal(x, cov).rvs()
    return u
```

The following is a simple implementation of the Metropolis–Hastings MCMC algorithm.

```
x0 = np.array([0, 0]) # initial starting point

chain = [x0] # chain stores all of the points visited

num_iterations = 10000
for i in range(num_iterations): # iterate the algorithm

    x = chain[-1] # current point is last in the chain

    u = Q(x) # proposed point

    alpha = P(u)/P(x) # compute acceptance ratio

    u = np.random.uniform() # draw random number in range 0 to 1

    if u<alpha: # accept the proposed point with probability
        x_new = u
    else: # reject the proposed point
        x_new = x

    chain.append(x_new) # append new point to the chain
```

The result of the algorithm is a Markov chain of points stored in the variable "chain". Let's look at the first 100 points in this chain, these are plotted in figure A.1. We can see that the chain executes a random walk across the 2-dimensional parameter space. The size of the steps in the random walk is governed by our choice of the proposal distribution.

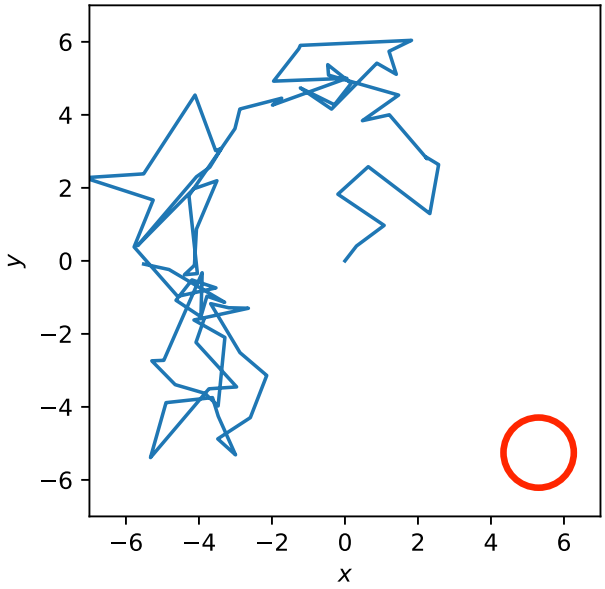

Figure A.1: The chain starts at the origin, where we initialised it. It immediately (in about 10 steps) heads outwards to a region of higher probability near a radius of $r_0 = 5$. Thereafter, it is possible for the chain to go to regions of lower probability, but there is a tendency for it to stay in regions of high probability. In fact, we can already see the chain begins to diffuse around the circle $x^2 + y^2 = r_0^2$. However, in these first 100 steps it doesn't get all the way around the circle; there is an excess of points at the 9 o'clock position. (The red circle in the lower right of the plot indicates the width of the proposal distribution.)

We can use the full chain to visualise the target distribution $P(x)$ by plotting a histogram. However, it is always a good idea to clean up the chain before plotting such histogram. The first few points in the chain may not be representative of the whole; this is because we choose the starting point arbitrarily and it might have been in a region of low probability (see figure A.1). Therefore it is common to remove some points from the start of the chain; this portion of the chain is called the *burn in* phase. After the burnin, we ideally want each point in the chain to be an independent random variable drawn from the target distribution. However this is clearly not the case for our chain; in figure A.1 we see that in the first 100 points there are none in over half of the circle. This is because the MCMC chain takes some time to move from place to place. Therefore, it is common to *thin* the chain by taking every n^{th} point. The choice of n should be guided by how long it takes the chain to move around the parameter space (what is known as the *autocorrelation length*). Here we choose to use a burnin length of 100 and a thinning of 50 (although this is probably

a bit too small).

The corner package[1] provides an easy way to plot multidimensional histograms, also known as *corner plots*, see figure A.2.

```
samples = np.array(chain)[100::50] # remove the burn in and perform the thinning

import corner
corner.corner(samples, labels=[r'$x$', r'$y$'])
```

The ring structure of the target distribution $P(x)$ can be clearly seen in figure A.2.

There are several potential problems that can arise with the Metropolis-Hastings MCMC algorithm. Some of these can be seen even in our above example

1. **Autocorrelation.** We want independent random samples from the target distribution. However, as we have seen it takes time for the MCMC chain to move around. Points in the chain separated by less than the autocorrelation length are not independent. If the autocorrelation length is L, then this means that to get N independent samples we have to run for $L \times N$ iterations which can take a long time. You might think of increasing the width of the proposal distribution to allow the chain to make bigger steps and hence move around faster decreasing the autocorrelation length. This can work, however it also runs into the acceptance problem...

2. **Acceptance.** At each iteration of the algorithm a new point u is proposed. If this point has a lower value of P than the current point then it is only accepted with probability $P(u)/P(x) < 1$ (if it has a higher value of P the is definitely accepted). If the proposal distribution keeps proposing points with low probability then it could take many iterations before our chain even moves a single step. A poor choice of proposal distribution typically leads to a low acceptance rate.

3. **Getting stuck.** Sometimes the target distribution can have multiple disconnected regions of high probability. (This was not the case is our example). For the chain to converge we need it to move backwards and forwards between these regions many times. This poses a serious problem for MCMC algorithms. When the chain is is one region it is very unlikely that our proposal distribution will propose a new point in the other. This can lead to our chain getting stuck in just one region and failing to adequately explore the full space.

[1] https://corner.readthedocs.io/en/latest/

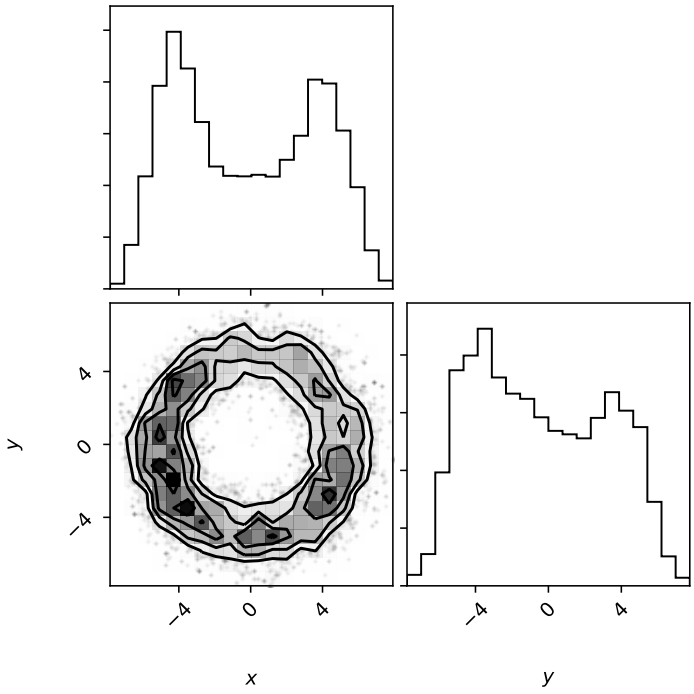

Figure A.2: An example of a *corner plot*, or *triangle plot* for the distribution $P(x)$. The lower left panel shows a 2-dimensional histogram of the point in our chain. The other two panels show the 1-dimensional marginalised histograms for the x (right) and y (top) dimensions. Our MCMC algorithm has performed fairly well. Although, it looks like we haven't run our MCMC for quite long enough. There seem to be more points near the 9 o'clock position than the rest of the ring. If we look back at figure A.1 we see that the chain initially went in this direction. We should probably run our algorithm for more iterations (say 100000) and apply more thinning to the resultant chain (say, take every 100^{th} point). If we did this we would expect to see this slight imbalance disappear.

4. **Serial.** The basic Metropolis-Hasting algorithm described does not lend itself well to being run in parallel.

5. **Doesn't evaluate the evidence.** The basic Metropolis-Hasting algorithm described can be used to draw random points from a posterior distribution. This allows us to estimate parameter values and find their uncertainties. However, it does not give us the Bayesian evidence.

Fortunately, a lot of research has been done into MCMC algorithms and this has resulted in variations to the Metropolis-Hastings algorithm designed to overcome some (all?) of these problems. There are many variations and implementations available and free to use. Implementing Metropolis-Hastings for yourself is a good way to learn about MCMC algorithms, but for serious research problems we should use a tried and tested package.

A.2 Variations of the MCMC algorithm

One such package is EMCEE[2]. This is a Python package that implements another type of MCMC algorithm (not Metropolis-Hasting) which is called *affine invariant Markov chain Monte Carlo ensemble sampling*. We will not describe this algorithm at all here[3], we will just briefly demonstrate how to use this package to sample from our ring distribution $P(x)$.

```
import emcee

ndim = 2 # our problem is 2 dimensional
nwalkers = 10 # instead of a single chain this algorithm uses several

x0 = np.random.randn(nwalkers, ndim) # choose random starting points

def log_P(x): # the algorithm needs the log probability function as an input
    return np.log(P(x))

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_P) # setup the sampler

state = sampler.run_mcmc(x0, 100) # run 100 burn in iterations

sampler.reset()

sampler.run_mcmc(state, 10000) # run algorithm for 10000 iterations

samples = sampler.get_chain(flat=True, thin=50) # get the thinned mcmc chain

corner.corner(samples, labels=[r'$x$', r'$y$'])
```

The histogram of the resulting chain, stored in "samples", is plotted in figure A.3.

[2] https://emcee.readthedocs.io/en/stable/

[3] For details, see Goodman, J., Weare, J., *Ensemble samplers with affine invariance*, Commun. Appl. Math. Comput. Sci. 5 , 1, 65-80 (2010)) doi:10.2140/camcos.2010.5.65.

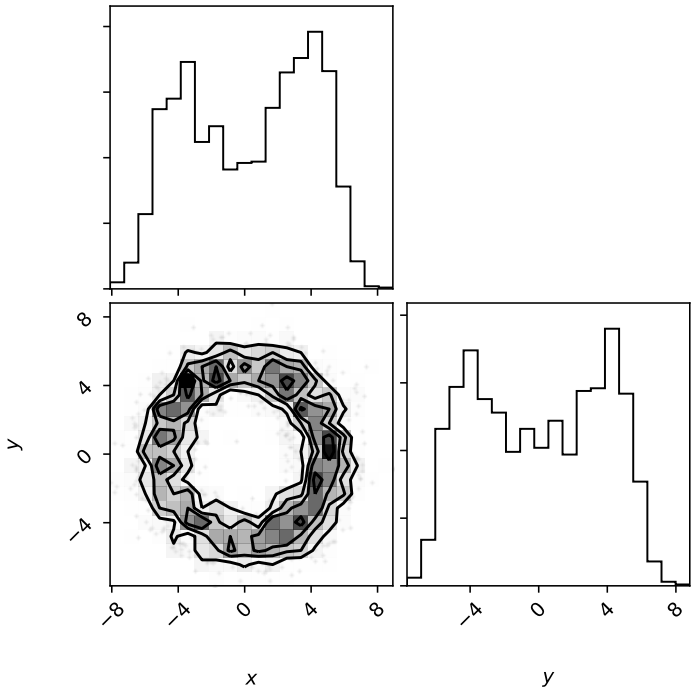

Figure A.3: The 2-dimensional histogram of the chain produced by the EMCEE algorithm. It should not be a surprise that for similar parameter to those used in figure A.2 these samples appear to be slightly better (there is no imbalance visible in the ring). The emcee algorithm includes several improvements over the "vanilla" Metropolis-Hastings MCMC algorithm implemented above.