In this exercise session you will implement a simple Metropolis MCMC sampler in your programming language of choice. You will test this sampler on a simple toy problem – drawing samples from a two-dimensional Gaussian – and then on two more realistic problems. The exercises are designed to take longer than the allotted 45 minutes so don't worry if you don't finish. Hopefully people with all levels of experience will learn something! If you have previous experience with MCMC and if you have already implemented a Metropolis sampler, why don't you take this time to implement a more sophisticated method (ensemble, Hamiltonian, nested, or otherwise), try out a new programming language, or help out other students with less experience.

Toy problem: a two-dimensional Gaussian In the following section you will implement your own Metropolis MCMC sampler and test it by drawing samples from a two-dimensional Gaussian but the first step is to implement the probability distribution as a function that takes in a two-dimensional vector θ and returns:

$$\ln p(\theta) \quad = \quad -\frac{1}{2}\,\theta^{\mathrm{T}}\,\Sigma^{-1}\,\theta + \text{constant} \tag{1}$$

where

$$\Sigma \quad = \quad \begin{pmatrix} 1.0 & -0.08 \\ -0.08 & 0.01 \end{pmatrix} \tag{2}$$

Note: you definitely want to compute $\ln p$ (not just p). *Why?*

To test your probability function, ensure that the difference between the function evaluated at two (random) points in parameter space is the same as what I get. For example, make sure that you find the following:

$$\ln p([0.5, -0.1]^{\mathrm{T}}) - \ln p([-0.01, 0.3]^{\mathrm{T}}) \quad \equiv \quad 11.80847\ldots \quad . \tag{3}$$

If you don't get this result on the first try, debug your function until you do.

A homegrown Metropolis sampler In this section, you will implement a Metropolis MCMC sampler to draw samples from the distribution defined in the previous section. As a reminder, the steps in a Metropolis MCMC are described here in words:

1. Initialize the parameters as $\theta(t = 0)$.
2. Propose an update $q = \theta(t) + \sigma\,\delta$ where σ is a tuning parameter and δ is drawn from a zero mean, unit variance Gaussian. Alternatively, you can try updating θ only one dimension.
3. Compute the acceptance probability $r = \min\left(1,\, \frac{p(q)}{p(\theta(t))}\right)$.
4. Draw u from a uniform distribution between 0 and 1. If $u < r$, accept the proposal and set $\theta(t + 1) = q$ to the chain. Otherwise, reject the proposal and set $\theta(t + 1) = \theta(t)$. Append $\theta(t + 1)$ to the chain. *Note: in every iteration of this procedure a new entry is added to the chain – even if the sample isn't accepted!*
5. Go to step 2 and repeat.

Here are some suggestions to keep in mind for your implementation:

- In this function, you should track the acceptance fraction. Keep track of how many proposals you accept.
- As mentioned above, don't compute or use $p(\theta)$ directly – you should only ever use the logarithm of this function.
- Your implementation must support parameter *vectors*. For this problem, we're working in 2-D but don't special case to that because it might be necessary to be able to sample more parameters later.
- Similarly, you should implement the sampler in such a way that it is easy to swap out a different probability function. Most programming languages allow you to pass function pointers so taking the log-probability function as an argument is probably the best interface.
- Finally, to start the sampler, you must choose an initial location in parameter space. Don't hard-code this location – take it as an argument – because we'll test different initialization methods later.

Implement the Metropolis MCMC method as a function with a calling sequence something like

```
samples, acc_frac = metropolis(log_prob_func, sigma, inital_theta, num_steps)
```

This function should return the chain of samples and the accumulated acceptance fraction.

Testing, tuning, and convergence (aka time to break your sampler) MCMC methods are notoriously hard to test rigorously so instead you can just test your implementation qualitatively for today. To start, initialize at a reasonable location (*What is a good way to choose this in general?*) and *run some chains and look at a few diagnostics*

- What is the acceptance fraction? Is this unreasonably high or low? Note: this is a very easy problem so the acceptance fraction will probably be higher than you would get in any problem in The Real World™.
- Compute the sample mean and covariance matrix of the chain. Is this close to what you would expect? What do you expect and why? What happens as you run the chain for longer?
- Plot the trace (value as a function of step number) for each parameter. See Figure 1 for an example of what to expect.
- Plot the scatterplot matrix or corner plot to visualize the covariances in the problem. In Python you can use corner.py[1] and other plotting libraries probably have similar functionality but, otherwise, you can just make a scatterplot of the chain values. Figure 2 shows an example using corner.py.
- Try changing σ and your initialization and remake these plots to see what happens. What happens as you make σ extremely large 10^4 or small 10^{-4}? What happens if you move the initial guess far away from the peak of the distribution?

[1] http://corner.readthedocs.io

As discussed in lecture, the best quantification of sampler performance is the integrated autocorrelation time $\tau_{\rm int}$.[2] This provides an estimate of the number of steps of MCMC required to obtain an independent sample. Since MCMC is used to compute integrals, an estimate of $\tau_{\rm int}$ is required to compute the sampling uncertainty on the approximation

$$\int f(\theta)\, p(\theta)\, \mathrm{d}\theta \quad \approx \quad \frac{1}{N}\sum_{n=1}^{N} f(\theta^{(n)}) \quad \text{where}\, \theta^{(n)} \sim p(\theta) \tag{4}$$

and the sampler that produces a chain with a smallest value of $\tau_{\rm int}$ at fixed computational cost is the best.

Write or find a function that computes the autocorrelation function[3] *of a one-dimensional chain and compute this for a few choices of* $f(\theta)$. Figure 3 gives a few suggestions for functions to consider. *Then, find or implement a function that estimates the integrated autocorrelation time*:

$$\tau_{\rm int,est} \quad \approx \quad C(0) + 2\sum_{\tau=1}^{W} C(\tau) \tag{5}$$

where $C(\tau)$ is the autocorrelation function and W is a tuning parameter. Note: autocorrelation time estimates are always very noisy and you need to run a long chain to get a reliable estimate. Furthermore, the choice of W can be subtle. Try several values of W or read A. Sokal's notes[4] on how to choose W more robustly. For more information about autocorrelation analysis, including Python code, take a look at this blog post: `https://dfm.io/posts/autocorr/`.

For $\sigma = 0.1$, I find integrated autocorrelation times of 577 steps and 383 steps for $f(\theta) = \theta_1$ and $f(\theta) = \theta_2$ respectively. *Do you find the same? If not, why not? Try changing* σ *to see how the autocorrelation function and times change. Can you find a better choice of* σ*? What if you use different values of* σ *for each parameter? How many tuning parameters are there in this method?*

Bonus: Try running a different sampling algorithm on the same problem and compare performance using an estimate of the autocorrelation time. For example, without any tuning, emcee[5] gets autocorrelation times of about 31 steps for both $f(\theta) = \theta_1$ and $f(\theta) = \theta_2$. Can you match that performance using Metropolis? What changes did you have to make?

---

[2] Note: there isn't just *one* autocorrelation time. $\tau_{\rm int}$ will, in general, be different for every different $f(\theta)$ in Equation 4. [3] If you write your own function, don't directly sum the variance at each lag – this will be way too slow. Instead, you could try using an FFT. [4] See pages 15–16 of this note: `http://www.stat.unc.edu/faculty/cji/Sokal.pdf` [5] `http://emcee.readthedocs.io`

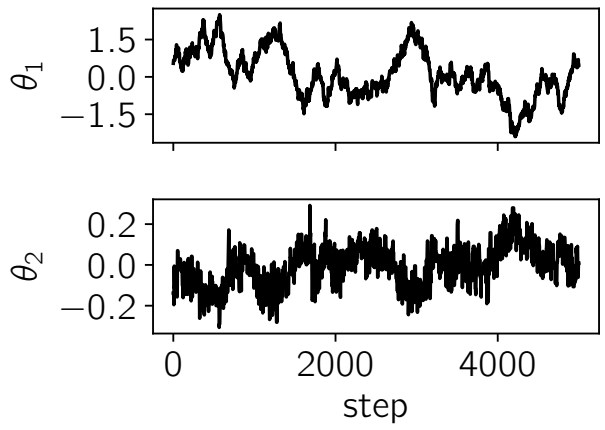

Fig. 1.— The parameter traces for Metropolis MCMC sampling of a two-dimensional Gaussian. The plots show the parameter values as a function of step number in the chain. To show the autocorrelation on this plot, I zoomed in to only show the first few steps but the full chain used for the following figures had 4×10^5 steps.

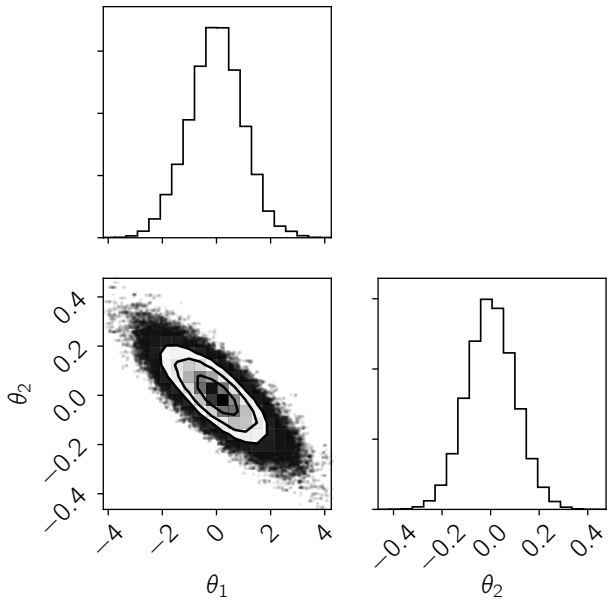

Fig. 2.— A scatterplot matrix or corner plot of the chain from Figure 1. The central panel shows the scatterplot of the parameter values from the chain and the histograms show the corresponding projections. The scatterplot shows the Monte Carlo estimate of the joint probability density $p(\theta_1, \theta_2 \,|\, \mathrm{data})$ and the histograms are estimates of the marginalized probability densities $p(\theta_1 \,|\, \mathrm{data})$ and $p(\theta_2 \,|\, \mathrm{data})$.

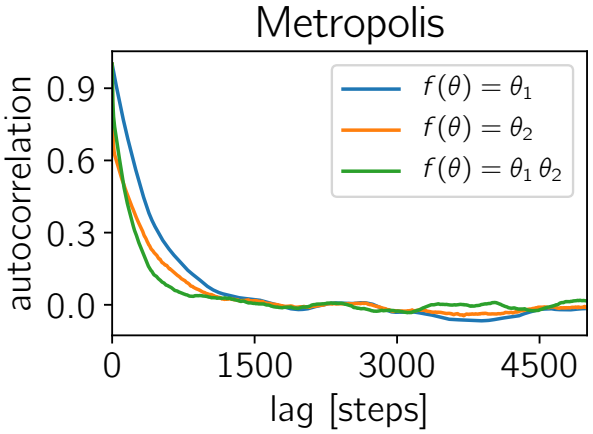

Fig. 3.— The autocorrelation function for chains of $\{f_k(\theta^{(n)})\}_{n=1}^N$ for different choices of f_k. Compare this to Figure 1 to see how the autocorrelation relates to the chain behavior. Why do the autocorrelation functions have different scales? Is the sampler well tuned for $f(\theta) = \theta_1$?

A more realistic problem: fitting a line to data In this section, you will fit a line to the small data set plotted in Figure 4 and available online at `https://github.com/dfm/ahw2018/blob/master/data.txt`[6]. If you just have data points with Gaussian uncertainties and Gaussian or broad priors, you probably shouldn't use MCMC (because the posterior can be sampled analytically) so let's make it a little more interesting: there is an intrinsic scatter in the relationship and there are uncertainties on the x values.

We'll start by ignoring the x uncertainties but including the intrinsic scatter parameterized by the logarithm of the scatter $\ln s$. In this case, the likelihood is:

$$\ln p(y \,|\, m,\, b,\, \ln s) \quad = \quad -\frac{1}{2}\sum_{n=1}^{N}\left[\left(\frac{y_n - m\,x_n - b}{\sigma^2 + s^2}\right)^2 + \ln(2\,\pi\,(\sigma^2 + s^2))\right] \tag{6}$$

(this derivation is left as an exercise) and a reasonable choice of "uninformative" prior is[7]:

$$\ln p(m,\, b,\, \ln s) \quad \propto \quad -\frac{3}{2}\ln(1 + b^2) + B(m,\, b,\, \ln s) \tag{7}$$

where $B(m,\, b,\, \ln s)$ is a finite constant when m, b, and $\ln s$ take values in a "reasonable range" (How should you choose this in general?) and $-\infty$ otherwise.

Implement this model in code and try sampling from it using the sampler that you implemented previously or another package. Make all the diagnostic plots from above but include one new plot: the posterior predictive distribution (see Figure 5). To achieve this, plot the line predicted for a few samples randomly selected from the chain on top of the observed data.

Finally, implement the full model where uncertainties in x are also included. To do this, you will need to choose a prior on $x_{\rm true}$ – maybe something like $\mathcal{U}(0,\, 10)$ – and sample the 13-dimensional problem instead of the 3-dimensional case from before. Before starting on this, you should draw the graphical model and consult with any other students who are working on this part.

---

[6] In this file, the columns are x, y, σ_y, and σ_x. [7] See J. VanderPlas's blog for a discussion of this choice: `http://jakevdp.github.io/blog/2014/06/14/frequentism-and-bayesianism-4-bayesian-in-python/`.

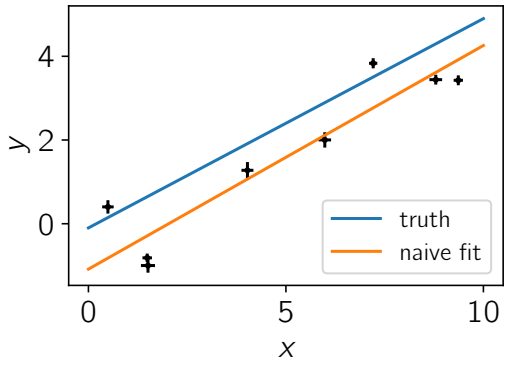

Fig. 4.— The black points with error bars show the simulated data used for this experiment. The blue line is the true underlying model that generated the data and the orange line is what you get if you run linear least squares and ignore the intrinsic scatter in the relation.

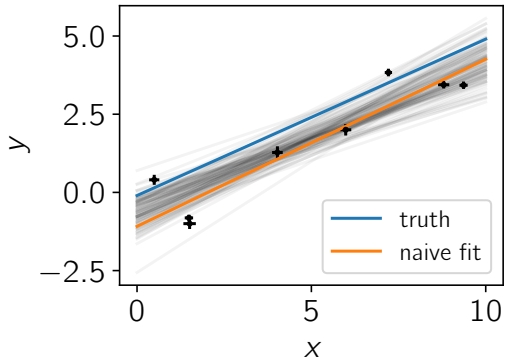

Fig. 5.— Like Figure 4 but the gray lines show the prediction for 100 random samples from the chain. The uncertainty is larger than for the least squares result but the true model is within the credible region.

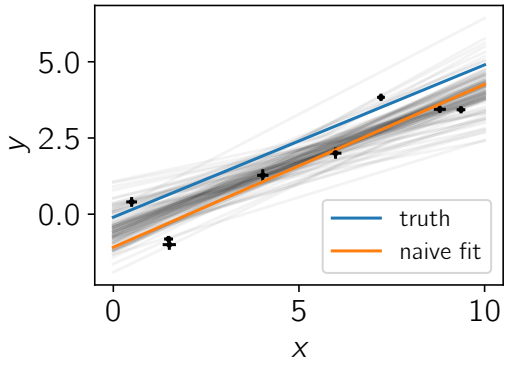

Fig. 6.— Like Figure 6 but taking the x uncertainties into account.