The Banana Distribution

Biostats 213 Final

Teia Noel

December 12, 2018

1 Introduction

In this project, I explore a method for sampling from a banana distribution, a two-dimensional probability distribution defined as follows:

$$\pi(x_1, x_2) \propto e^{\frac{-x_1^2}{2}} e^{\frac{-(x_2-2(x_1^2-5))^2}{2}}; x_1, x_2 \in \mathbb{R} \tag{1}$$

Directly sampling from the banana distribution would entail first generating a sampling of x_1 from the marginal, $\pi(x_1)$, followed by sampling x_2 from the conditional, $\pi(x_2|x_1)$, or vice versa. However, in this case, the marginals, $\pi(x_1)$ and $\pi(x_2)$ do not have closed form solutions. Here, I conduct Markov chain Monte Carlo (MCMC) sampling, a class of techniques useful in sampling from complicated multidimensional distributions such as this one.

2 Methodology

2.1 The Metropolis Hastings Algorithm

Using Python3, I implemented the Metropolis Hastings (MH) approach for MCMC sampling:

```python
import numpy as np
from numpy.random import uniform as unif
from matplotlib import pyplot as plt
import math
import pymc as pm
```

```python
# Samples from banana dist
def banana(x): # x is a 2d array
    x1=x[0]
    x2=x[1]
    p = np.exp(-(x1**2)/2)*np.exp(-((x2-2*(x1**2-5))**2)/2)
    return p
```

```
def metropolis(n,w,burnin): # takes in number of iterations, n, jump size, w, and burn-in
    vals = [] # stores accepted draws from distribution
    x=np.array([0.0,0.0])
    vals.append([x[0],x[1]])

    for i in range(1,n):
        jump1=unif(-w,w) # jump size for x1
        jump2=unif(-w,w) # jump size for x2
        x_next=np.array([x[0]+jump1,x[1]+jump2]) # candidate draw from distribution
        acceptance_ratio=min(1,banana(x_next)/banana(x))
        u=unif(0,1)

        if(u<acceptance_ratio):
            x=x_next # accept and store candidate draw
            vals.append([x[0],x[1]])

    return vals[burnin:] # return vals w/out burn-in
```

```
vals1 = metropolis(20000,0.5,8000) # run with jump size = 0.5
vals2 = metropolis(20000,1.0,8000) # jump size = 1
vals3 = metropolis(20000,4.0,1000) # jump size = 4
```

In this method, the metropolis function takes in the arguments n, the number of iterations the algorithm runs, w, the jump size for a candidate draw from the distribution, burnin, the desired number of accepted draws found to precede the stationary distribution and are thereby thrown away, and thin, the thinning factor. The function initiates an array, vals, that stores the accepted draws (x_1, x_2) from the distribution. Initially, it accepts an arbitrary (x_1, x_2), which I chose to be $(0, 0)$.

For the specified number of iterations, n, a candidate draw is calculated by drawing a random uniform value from U(-w,w) for each of x_1 and x_2, and adding these values to the previously accepted (x_1, x_2) draw. An acceptance ratio is determined by finding the minimum value between a choice of 1 or the ratio of the banana distribution evaluated at the candidate tuple to the banana distribution evaluated at the previously accepted tuple (these values are returned by the banana function). The candidate tuple is then accepted under the condition that a draw from U(0,1) is less than the acceptance ratio. The intuition is that the closer a candidate tuple and a previously accepted tuple are in value, the more likely the candidate will be representative of the distribution.

After a number of draws are accepted after n iterations, the amount of burnin specified is eliminated and the remaining m^{th} value is returned, where m is the thin parameter. For the purposes of this project, I tested the metropolis function for jump sizes 0.5, 1, and 4.

2.2 The Raftery-Lewis Test

Initially, MH was run for 20,000 iterations with burnin=0 and thin=1 for all step sizes (all accepted values were output by the algorithm). The Raftery-Lewis Test was run in order to gain insight on the number of iterations, the burn-in, and thinning factor necessary to reach convergence. The Raftery-Lewis function calls from the pymc package for MH paths generated from jump sizes=0.5, 1, and 4 are shown below. The function was called with a specified quantile to be estimated=0.025 and accuracy value for the quantile=0.01. Diagnostics for both x_1 and x_2 were generated. The maximum value for burn-in, thinning factor, and additional iterations were chosen amongst the diagnostics for both variables.

```
pm.raftery_lewis(vals1,q=0.025,r=0.01) # jump size = 0.5
pm.raftery_lewis(vals2,q=0.025,r=0.01) # jump size = 1.0
pm.raftery_lewis(vals3,q=0.025,r=0.01) # jump size = 4.0
```

2.3 Trace Plots

In order to judge whether or not my MH paths were well-mixed, I plotted the trace of x_1 and x_2 values over all iterations beyond the determined burn-in and with the determined thinning factor. If values seemed to get "stuck" in any region of the plot, it was deemed suspect of poor mixing. Otherwise, consistent oscillatory behavior indicated a well-mixed sample. The code for generating x_1 and x_2 trace plots is shown below for the MH path generated with jump size=0.5. The code was repeated with MH paths for jump sizes=1 and 4.

```
x1_vals1 = [item[0] for item in vals1] # extract x1 values from MH path
plt.plot(x1_vals1) # plot x1 values
plt.xlabel('iteration') # label x and y axes
plt.ylabel('x1')
plt.show() # show plot

x2_vals1 = [item[1] for item in vals1] # extract x2 values from MH path
plt.plot(x2_vals1) # plot x2 values
plt.xlabel('iteration') # label x and y axes
plt.ylabel('x2')
plt.show() # show plot
```

2.4 Geweke Convergence Test

The Geweke Test was was run in order to check if the MH paths converged to the stationary distribution. I used the pymc package's Geweke test, which, for a given MH path, takes the first 10% of the data points, partitions the last 50% of the data points, and calculates a z-score comparing the top 10% with each of the 20 partitions in the last 50%. The output is a plot of the 20 z-score values, and lines indicating bounds of 2 and -2 standard deviations. If a majority of the z-scores fall within this range, the MH path is determined to have converged. The function calls for the pymc Geweke test for MH paths generated with jump sizes 0.5, 1, and 4 are shown below.

```
scores1=pm.geweke(vals1) # jump size = 0.5
pm.Matplot.geweke_plot(scores1, "geweke1")

scores2=pm.geweke(vals2) # jump size = 1.0
pm.Matplot.geweke_plot(scores2, "geweke2")

scores2=pm.geweke(vals3) # jump size = 4.0
pm.Matplot.geweke_plot(scores3, "geweke3")
```

2.5 Plotting the Distribution

In order to visualize the resulting draws from the banana distribution, I generated a scatter plot for x_1 and x_2 values for the MH-generated distributions with jump sizes 0.5, 1, and 4. For the scatter plot corresponding to the distribution generated from jump size=0.5, I overlaid a trace of the first 200 iterations. Code for generating these plots is shown below for jump size=0.5, but similar code was used for the data points generated with jump sizes=1 and 4.

```
vals1_200=vals1[:200] # extract MH path for first 200 iterations
vals1_200_t=np.transpose(vals1_200) # transpose of 200 iterations
vals1_t=np.transpose(vals1) # transpose of all iterations

plt.plot(vals1_200_t[0],vals1_200_t[1],"r-",linewidth=0.5) # trace of first 200 iterations
plt.scatter(vals1_t[0],vals1_t[1],s=10) # scatter plot of all iterations
plt.xlabel('x1') # label x and y axes
plt.ylabel('x2')
plt.show() # display plot
```

3 Results

3.1 MH path generated with a jump size of 4 required the least amount of thinning

w=0.5

```
========================
Raftery-Lewis Diagnostic
========================

937 iterations required (assuming independence) to achieve 0.01 accuracy with 95 percent prob
ability.

Thinning factor of 11 required to produce a first-order Markov chain.

154 iterations to be discarded at the beginning of the simulation (burn-in).

48334 subsequent iterations required.

Thinning factor of 38 required to produce an independence chain.

========================
Raftery-Lewis Diagnostic
========================

937 iterations required (assuming independence) to achieve 0.01 accuracy with 95 percent prob
ability.

Thinning factor of 7 required to produce a first-order Markov chain.

63 iterations to be discarded at the beginning of the simulation (burn-in).

17171 subsequent iterations required.

Thinning factor of 22 required to produce an independence chain.
```

The Raftery-Lewis diagnostics for the MH path generated with jump size=0.5 indicated that a first-order Markov chain required a burnin of ∼160 and thinning factor of ∼15, with an additional ∼50,000 iterations.

w=1.0

```
========================
Raftery-Lewis Diagnostic
========================

937 iterations required (assuming independence) to achieve 0.01 accuracy with 95 percent prob
ability.

Thinning factor of 7 required to produce a first-order Markov chain.

168 iterations to be discarded at the beginning of the simulation (burn-in).

48111 subsequent iterations required.

Thinning factor of 33 required to produce an independence chain.

========================
Raftery-Lewis Diagnostic
========================

937 iterations required (assuming independence) to achieve 0.01 accuracy with 95 percent prob
ability.

Thinning factor of 2 required to produce a first-order Markov chain.

16 iterations to be discarded at the beginning of the simulation (burn-in).

4182 subsequent iterations required.

Thinning factor of 6 required to produce an independence chain.
```

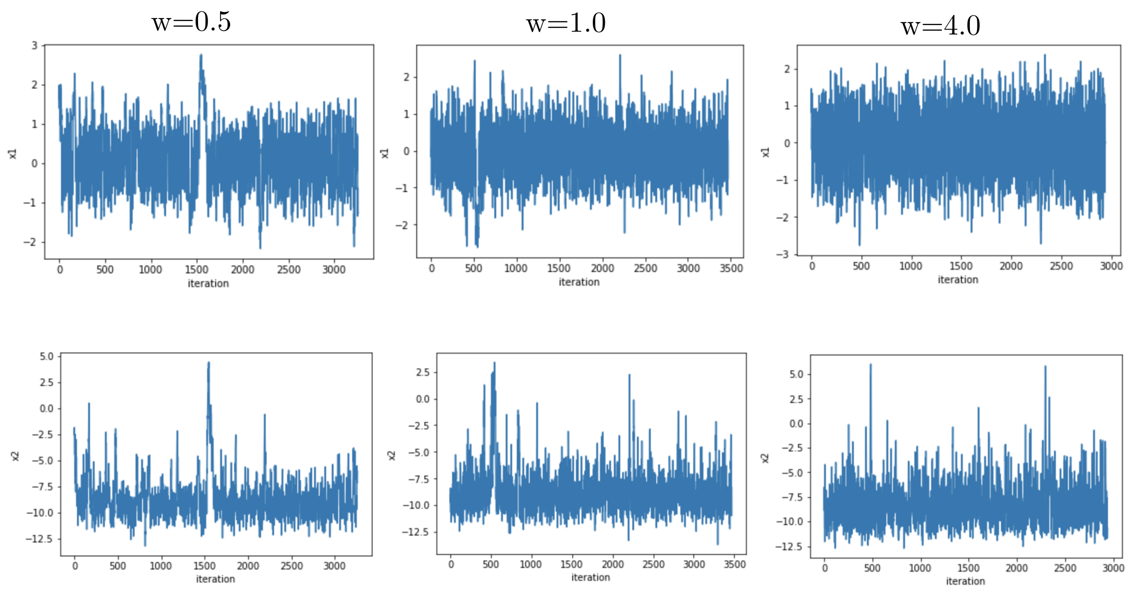
The Raftery-Lewis diagnostics for the MH path generated with jump size=1 indicated that a first-order Markov chain required a burnin of ∼170 and thinning factor of ∼10, with an additional ∼50,000 iterations.

w=4.0

```
========================
Raftery-Lewis Diagnostic
========================

937 iterations required (assuming independence) to achieve 0.01 accuracy with 95 percent prob
ability.

Thinning factor of 1 required to produce a first-order Markov chain.

6 iterations to be discarded at the beginning of the simulation (burn-in).

1788 subsequent iterations required.

Thinning factor of 3 required to produce an independence chain.

========================
Raftery-Lewis Diagnostic
========================

937 iterations required (assuming independence) to achieve 0.01 accuracy with 95 percent prob
ability.

Thinning factor of 1 required to produce a first-order Markov chain.

4 iterations to be discarded at the beginning of the simulation (burn-in).

1155 subsequent iterations required.

Thinning factor of 2 required to produce an independence chain.
```

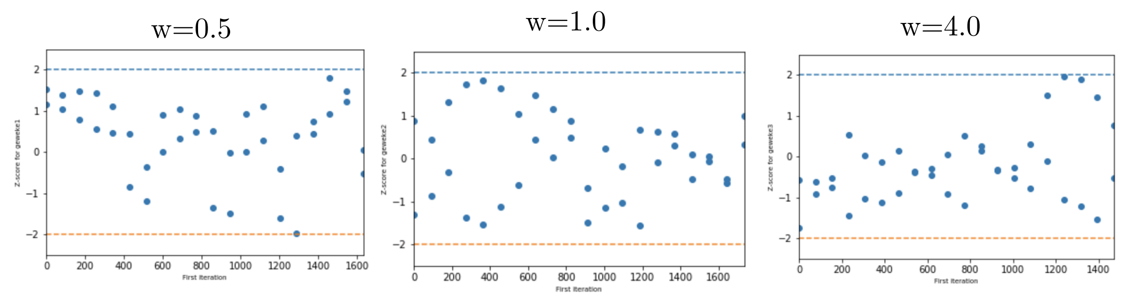
The Raftery-Lewis diagnostics for the MH path generated with jump size=4 indicated that a first-order Markov chain required a burnin of ∼10 and thinning factor of ∼1, with an additional ∼2,000 iterations.

3.2 MH path corresponding to a jump size of 4 produced the most well-mixed samplings



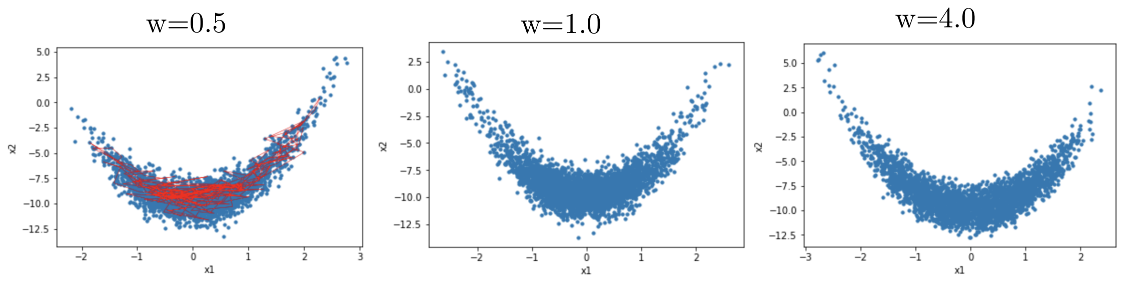
Trace plots for x_1 (first row) and x_2 (second row) showed samplings getting "stuck" mid-trace plot for a jump size of 0.5, and towards the beginning of the trace plot for a jump size of 1. A jump size of 4 produced a trace plot with the most consistent mixing.

3.3 Distributions resulting from all jump sizes converged



Geweke tests for distributions corresponding to all three jump sizes produced z-scores largely within the two-standard deviation range, indicating that all three distributions converged.

3.4 All jump sizes resulted in banana-shaped distributions



Distributions generated with all three jump sizes produced distributions that appear to follow the half-smile, banana-shaped distribution, as the name suggests. The trace of the first 200 iterations of the MH path corresponding to a jump size of 0.5 indicates that these samplings, without our determined burn-in amount, are well-mixed.

4 Conclusion

Eliminating burn-in values and implementing thinning based on diagnostics suggested by the Raftery-Lewis Test, the MH algorithm run with jump sizes=0.5, 1, and 4 all yielded promising samplings from our desired banana distribution. This conclusion is supported by the trace plots that suggest largely nice mixing, Geweke tests that indicate convergence, and x-y plots that illustrate the shape of our distribution of interest. However, I conclude that running the MH algorithm with the largest jump size of 4 enabled me to run the algorithm to produce a stable distribution in the least-costly manner. This is because the Raftery-Lewis diagnostics suggested the smallest burnin, thinning factor, and subsequently the fewest number of iterations necessary to achieve a stable sampling.

In the process of deciding on an MCMC algorithm to generate the banana distribution, I also considered implementing Gibbs sampling, wherein one iteratively samples from approximate conditional distributions if they yield simple forms. I found that $\pi(x_2|x_1) \propto N(\mu = 2(x_1^2 - 5), \sigma = 1)$. However, $\pi(x_1|x_2)$ did not resemble a simple distribution. One possibility would be to approximate this conditional using MH and the x_2 value approximated from $\pi(x_2|x_1)$. However, this is likely less desirable than the MH algorithm presented in this project, because every iteration of Gibbs would require thousands of iterations of MH to obtain a sampling for $\pi(x_1|x_2)$.

As mentioned earlier to motivate the usage of MCMC algorithms to begin with, direct sampling requires us to obtain $\pi(x_1)$ and $\pi(x_2)$, which do not have closed-form solutions. I thereby conclude that the MH algorithm illustrated here is our best option to simulate a sampling from the banana distribution.