Statistical Mechanics: HW 6

Truman Ellis

May 4, 2011

Our goal is to use umbrella sampling and Metropolis Monte Carlo to estimate the ratio $P(q_{\text{lowest minimum}})/P(q_{\text{barrier}})$ for the Muller potential. We do this by tracing out the path shown in red in Figure 1. At each point along the path, we define a biasing potential

$$V(x,y) = k([(x,y) - q_i] \cdot e_{q_i})^2$$

where q_i is the sample point, e_{q_i} is the tangent to the sample curve at that point, and k is the strength of the biasing potential. Then we run a small Metropolis Monte Carlo simulation starting at q_i with the Muller plus biasing potential. Within MMC simulation, we chose a maximum window size of $10^{-2} \times 10^{-2}$. This produced reasonable acceptance rates. We chose a biasing strength of 5000, which effectively channeled each simulation, but allowed enough overlap between simulations to glue all of the simulations together. After each mini simulation, I had an array of positions that the particle had visited. I then calculated the signed distance away from the reaction coordinate for each data point. With 2000 steps per simulation, this data fell into a nice Gaussian distribution (for high enough k). My first approach was to compute a histogram of this data then take the log and add βV. The challenge with this approach was finding an adequate way of gluing together the data from each of the mini simulations. After some further thought, I decided to make use of the fact that the data followed such a nice Gaussian distribution. I instead fit the data to a normal distribution via its mean and standard deviation. Taking the log of this and adding βV gave me a function that I could evaluate anywhere along the line. By evaluating two successive functions at the half nodes, I could calculate a shifting constant and vertically shift the second function in order to match the values.

I ran several simulations at varying sample densities, but 3200 sample points appeared to produce plots that lined up nicely, as shown in Figures 1 and 2. By subtracting the minimum value from the maximum, we get

$$\log(P(q_{\text{lowest minimum}})) - \log(P(q_{\text{barrier}})) = \log\left(P(q_{\text{lowest minimum}})/P(q_{\text{barrier}})\right) .$$

Then if we raise e to this value, we get the desired ratio of probabilities. For the simulation shown below, I got

$$\log(P(q_{\text{lowest minimum}})/P(q_{\text{barrier}})) = 578.42$$

or

$$P(q_{\text{lowest minimum}})/P(q_{\text{barrier}}) = e^{578.41} \approx 1.6e + 251 .$$

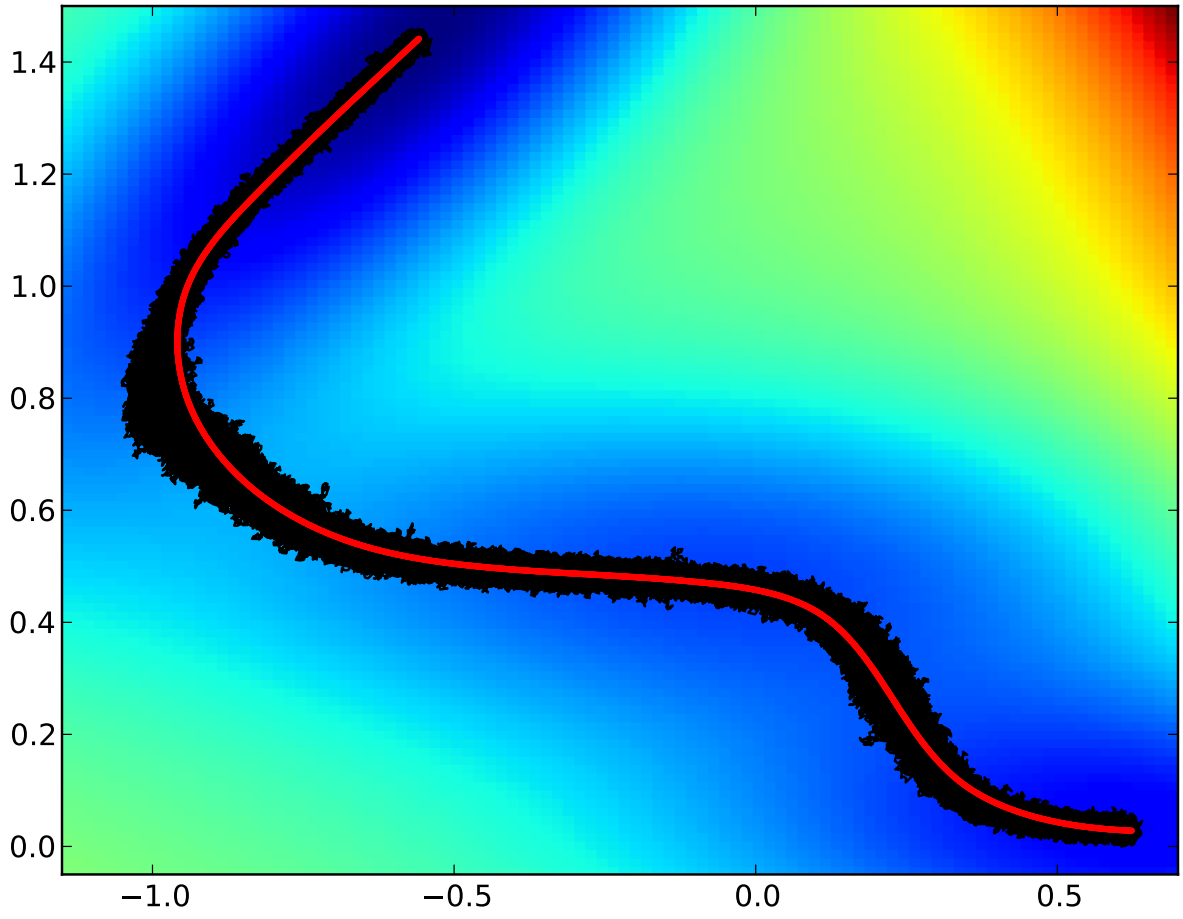

Figure 1: Sample path for umbrella sampling with mini Metropolis Monte Carlo simulations at each point

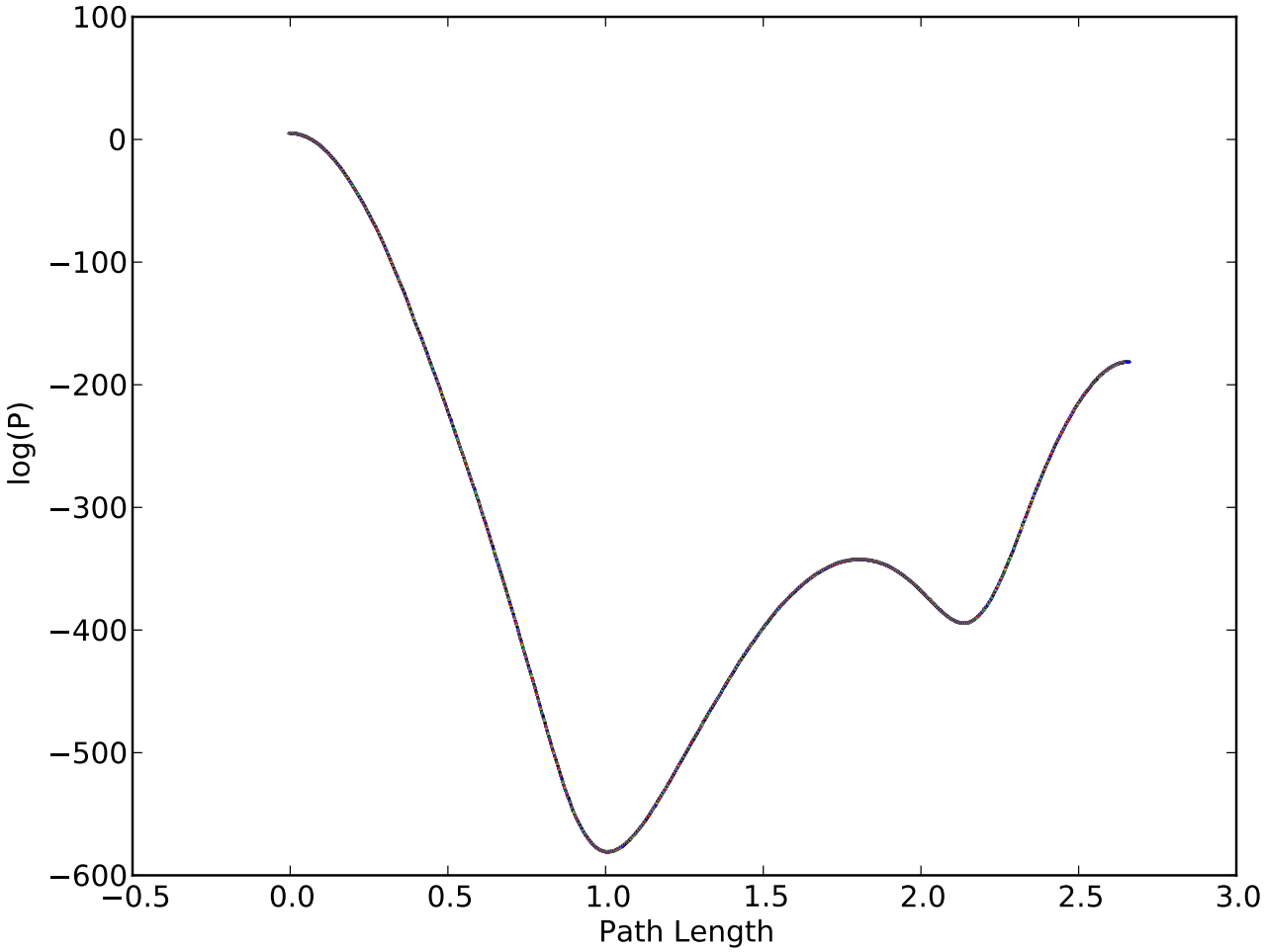

Figure 2: Log of the probability evaluated along the sample path

HW6.py

```
# Statistical Mechanics Homework 6
# Spring 2011
# Written by Truman Ellis

from pylab import *
close('all')

def gauss1(x, y):
    return -200*exp(-(x-1)*(x-1)-10*y*y)

def gauss2(x, y):
    return -100*exp(-x*x-10*(y-.5)*(y-.5))

def gauss3(x, y):
    return -170*exp(-6.5*(x+.5)*(x+.5)+11*(x+.5)*(y-1.5)-6.5*(y-1.5)*(y-1.5))

def gauss4(x, y):
    return 15*exp(.7*(x+1)*(x+1)+.6*(x+1)*(y-1)+.7*(y-1)*(y-1))

def muller((x, y)):
    return gauss1(x,y) + gauss2(x,y) + gauss3(x,y) + gauss4(x,y)

def Vadd((x,y), k, qi, e):
    return k*dot((x,y)-qi,e)**2

def potential((x,y), k, qi, e):
    return muller((x,y)) + Vadd((x,y), k, qi, e)

def gaussian(x, mu, s):
    return exp(-(x-mu)**2/(2*s**2))/sqrt(2*pi*s**2)

SamplePath = loadtxt("RefPath3200.txt")
Nsamples = SamplePath.shape[0]

# Initialize particle
m = 1.
beta = 10
k = 5000
Nsteps = 2000
maxstep = 5e-3

# Compute background pseudocolor plot of Muller potential
nx = 101
ny = 103
mp = zeros( (nx,ny) )
xmin = -1.15
xmax = 0.7
ymin = -.05
ymax = 1.5
x = linspace(xmin, xmax, nx)
y = linspace(ymin, ymax, ny)
for i in range(0,nx):
    for j in range(0,ny):
        mp[i,j] = muller((x[i],y[j]))

figure(1)
pcolor(x,y,mp.T)
hold(True)

# Initialize conditions for umbrella sampling
X = zeros((Nsteps,2))
D = zeros(Nsteps)
mu = zeros(Nsamples)
s = zeros(Nsamples)
PathLength = [0]
totalShift = 0
minLogP = inf
maxLogP = -inf
for i in range(0,Nsamples):
    X0 = SamplePath[i]
    # Calculate distance along path
    if i > 0:
        PathLength.append( PathLength[i-1] + norm(X0-SamplePath[i-1]) )
    # Calculate eq
    if (i == 0):
        e = (SamplePath[1]-SamplePath[0])/norm(SamplePath[1]-SamplePath[0])
    elif (i == Nsamples-1):
        e = (SamplePath[-1]-SamplePath[-2])/norm(SamplePath[-1]-SamplePath[-2])
    else:
        e = (SamplePath[i+1]-SamplePath[i-1])/norm(SamplePath[i+1]-SamplePath[i-1])
    U0 = potential(X0, k, X0, e)

    X[0] = X0
    Naccepted = 0.
```

```
    # Perform Metropolis Monte Carlo
    for n in range(1,Nsteps):
        Xn = X[n-1] + maxstep*2*(0.5-rand(2))
        Un = potential(Xn, k, X0, e)
        Paccept = min(1,exp(-beta*(Un-U0)))
        if rand() < Paccept:
            X[n] = Xn
            Naccepted += 1
        else:
            X[n] = X[n-1]
        D[n] = dot(e, X[n]-X0)

        U0 = Un

    # Compute mean and standard deviation
    mu[i] = mean(D) + PathLength[i]
    s[i] = std(D)
    # Shift curve to meet previous one
    if i > 0:
        x_shared = 0.5*(PathLength[i-1] + PathLength[i])
        DlogP = log(gaussian(x_shared, mu[i], s[i]) )\
                - log(gaussian(x_shared, mu[i-1], s[i-1])) \
                + beta*k*((PathLength[i]-x_shared)**2 - (PathLength[i-1]-x_shared)**2)
        totalShift += DlogP
    else:
        maxLogP = min(y)

    # Plot curves
    figure(2)
    x = PathLength[i] + linspace(-.005, .005,100)
    y = log(gaussian(x, mu[i], s[i])) + beta*k*(x-PathLength[i])**2 - totalShift
    minLogP = min(minLogP, min(y))
    plot(x,y)
    xlabel('Path Length')
    ylabel('log(P)')

    figure(1)
    plot(X[:,0],X[:,1],'k', linewidth=1)
    axis([xmin,xmax,ymin,ymax])

    print 'Sample point: '+str(i+1)+'/'+str(Nsamples)

# Plot and report results
figure(1)
plot(SamplePath[:,0], SamplePath[:,1], 'ro', markeredgecolor='r', markersize=2)
axis('tight')
show()

#Calculate acceptance rate
AccRate = Naccepted/Nsteps
print 'Acceptance Rate: %.3g'%(AccRate,)

# Calculate the probability ratio
P_ratio = exp(maxLogP - minLogP)
print 'Probability Ratio: %.3g'%(P_ratio,)
```