**ENGN2020 – HOMEWORK9**

### Problem 1

### Part (a)

The code of Metropolis algorithm is shown as below.

﻿def metropolis(initial\_guess = 1.0, steps = 500,maxStep = 0.1):

accept = 0

T = 300

kb = 1.38064852e-23

evToJ = 1.60218e-19

l\_old = initial\_guess

result = np.zeros((steps))

means = np.zeros((steps))

stds = np.zeros((steps))

acceptNums = np.zeros((steps))

for i in range(steps):

#get energy of old l

El\_old = get\_energy(l\_old)

#random jump to a new l

l\_new = l\_old+np.random.uniform(-1,1)\*maxStep

#get the energy of new l

El\_new = get\_energy(l\_new)

#calculate r by two energy, note that r is for logP

r = -(El\_new-El\_old)\*evToJ/kb/T

#if P(new)>P(old)

if r>0:

#save new l

accept = accept + 1

l\_old = l\_new

result[i] = l\_new

else:

#generate a new random value from [0,1]

prob = np.random.uniform(0,1)

#see if the we accept the new step

if math.exp(r) > prob:

accept = accept + 1

l\_old = l\_new

result[i] = l\_new

else:

result[i] = l\_old

#get min

current = result[:i+1]

current\_mean = np.mean(current)

means[i] = current\_mean

#get std

current\_std = np.std(current)

stds[i] = current\_std

#get accept ratio

acceptNums[i] = accept/(i+1)

return result,means,stds,acceptNums

Choose 0.5 Å as the start point, and set T by 300K and max step size by 0.1 Å.

The figure of average of ***l*** versus step number is show in Fig.1.



**Fig 1.** The figure of average of l versus step number

The figure of standard deviation of ***l*** versus step number is show in Fig.2.



**Fig 2.** The figure of standard deviation of l versus step number

The figure of fraction of steps accepted versus step number is show in Fig.3.



**Fig 3.** The figure of standard deviation of l versus step number

The histogram of ***l*** after 500 steps is shown in Fig.4.



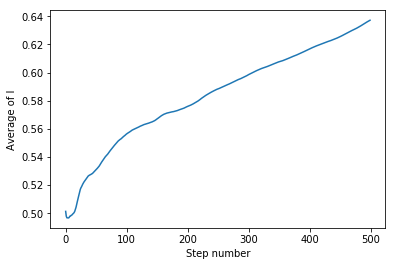
**Fig 4.** The figure of standard deviation of l versus step number

From the histogram, we can find that samples distribute normally at nearly 0.75 Å, where the corresponding energy is the smallest.

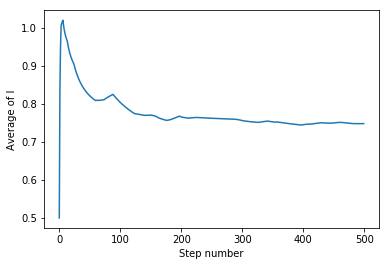
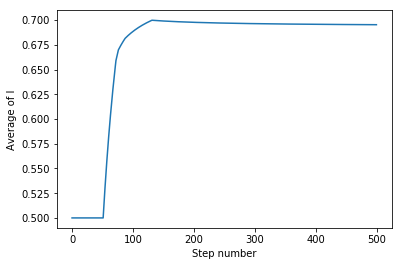
### Part (b)

Under different maximum step size, namely 0.01 Å, 0.1 Å, 1 Å, 10 Å, the plots of average of l against step number is shown in Fig.5.

As the maximum step size increases, the algorithm converges more quickly. However, when the step size is too big, sometimes, the algorithm couldn`t converge to the right result. The system becomes instable.



**(a) (b)**



**(c) (d)**

**Fig 5.** Plots of average of l against step number under different maximum step sizes. **(a)**0.01 Å**(b)**0.1 Å**(c)**1 Å**(d)**10 Å

0.1 Å seems a very optimal maximum step size, the algorithm converges quite fast, and according to my experiments, when the initial guess is between 0.3 Å to 2 Å, every time the algorithm can converge to the same result.

### Part (c)

### Problem 2

Where:

To obtain a linear model, we take the base-10 logarithm:

Based on the given data, the base-10 logarithm is shown in Table 1.

**Table.1** The data after base-10 logarithm

|  |  |  |
| --- | --- | --- |
| Nu | Re | Pr |
| 0.29393681 | 0 | -0.1366771 |
| -0.0464336 | -1 | -0.1366771 |
| -0.3704885 | -2 | -0.1366771 |
| 0.39963911 | 0 | 0.17609126 |
| 0.06149018 | -1 | 0.17609126 |
| -0.2580609 | -2 | 0.17609126 |

By following the tutorial of the ***sampyl*** library, the parameter vector can be calculated by the following code.

﻿import sampyl as smp

from sampyl import np

X = np.array([[1, 0, -0.13667714],

[1, -1, -0.13667714],

[1, -2, -0.13667714],

[1, 0, 0.176091259],

[1, -1, 0.176091259],

[1, -2, 0.176091259]])

y = np.array([[0.293936814],

[-0.046433586],

[-0.370488466],

[0.399639115],

[0.061490177],

[-0.258060922]])

def logp(b, sig):

#define the model

model = smp.Model()

# Predicted value

y\_hat = np.dot(X, b)

# Log-likelihood

model.add(smp.normal(y, mu=y\_hat, sig=sig))

# log-priors

model.add(smp.exponential(sig),

smp.normal(b, mu=0, sig=100))

return model()

start = smp.find\_MAP(logp, {'b': np.ones(3), 'sig': 1.})

nuts = smp.NUTS(logp, start)

chain = nuts.sample(2100, burn=100)

import matplotlib.pyplot as plt

plt.plot(chain.b)

A chain of 2000 samples is calculated, and the figure of those samples is shown in Fig.5.



The histograms of each parameter in vector b is shown in Fig.6.



The code below is used to calculate the 95% confidence section.

﻿

import numpy as np

import scipy.stats

def mean\_confidence\_interval(data, confidence=0.95):

a = 1.0 \* np.array(data)

n = len(a)

m, se = np.mean(a), scipy.stats.sem(a)

h = se \* scipy.stats.norm.ppf((1 + confidence) / 2., n-1)

return m, m-h, m+h

After calculation, the 95% confidence section of is [0.00976, 0.01621].

The 95% confidence section of is [-0.00281,0.00236]

The 95% confidence section of is [-0.0199,0.01055]