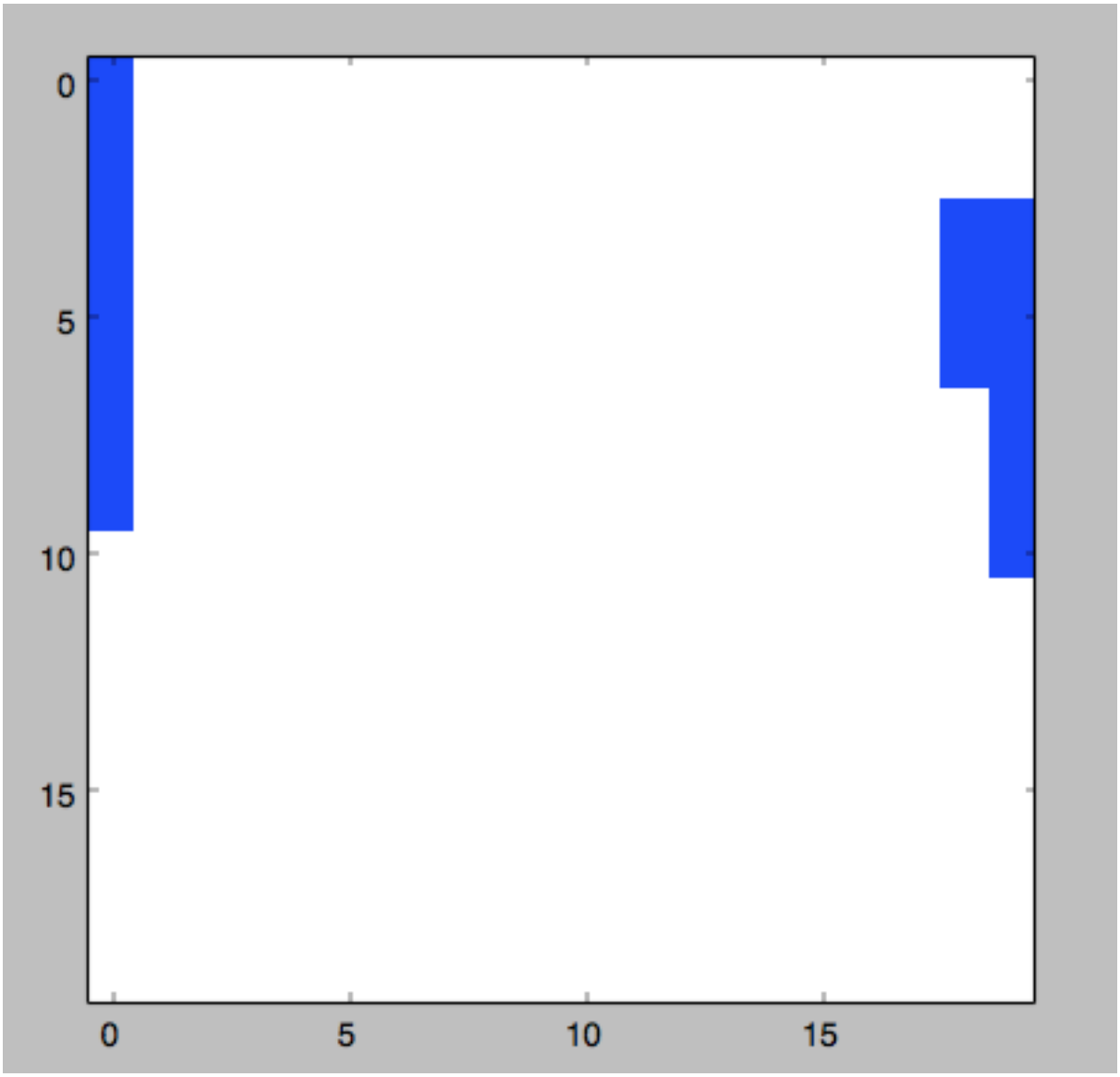
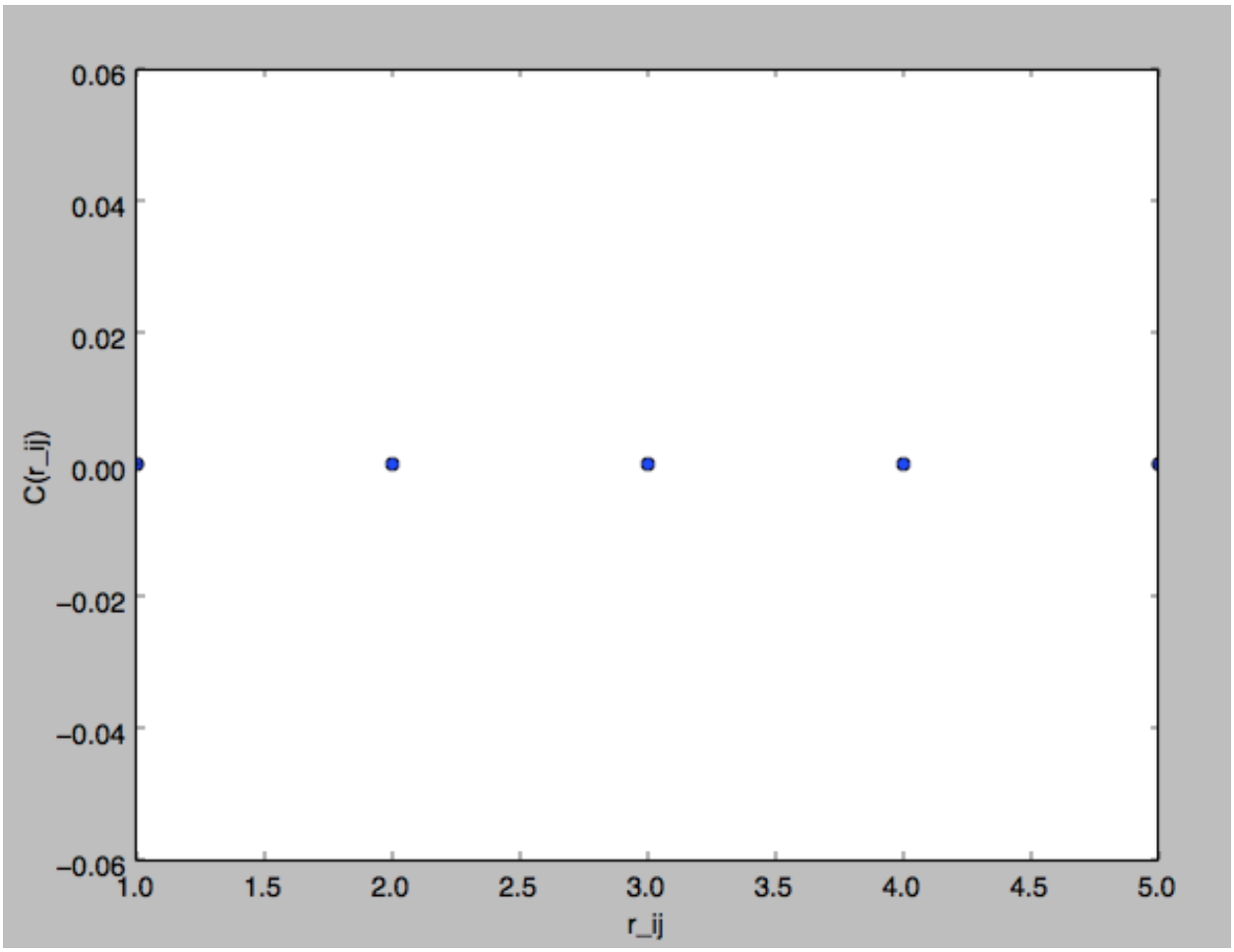
Problem Set 11 Extra Credit Questions
Alexander Brandt
CHEM 220A
12/16/2013

1. Ising Model in a Coordinate Biased Potential (IMSM 6.10)

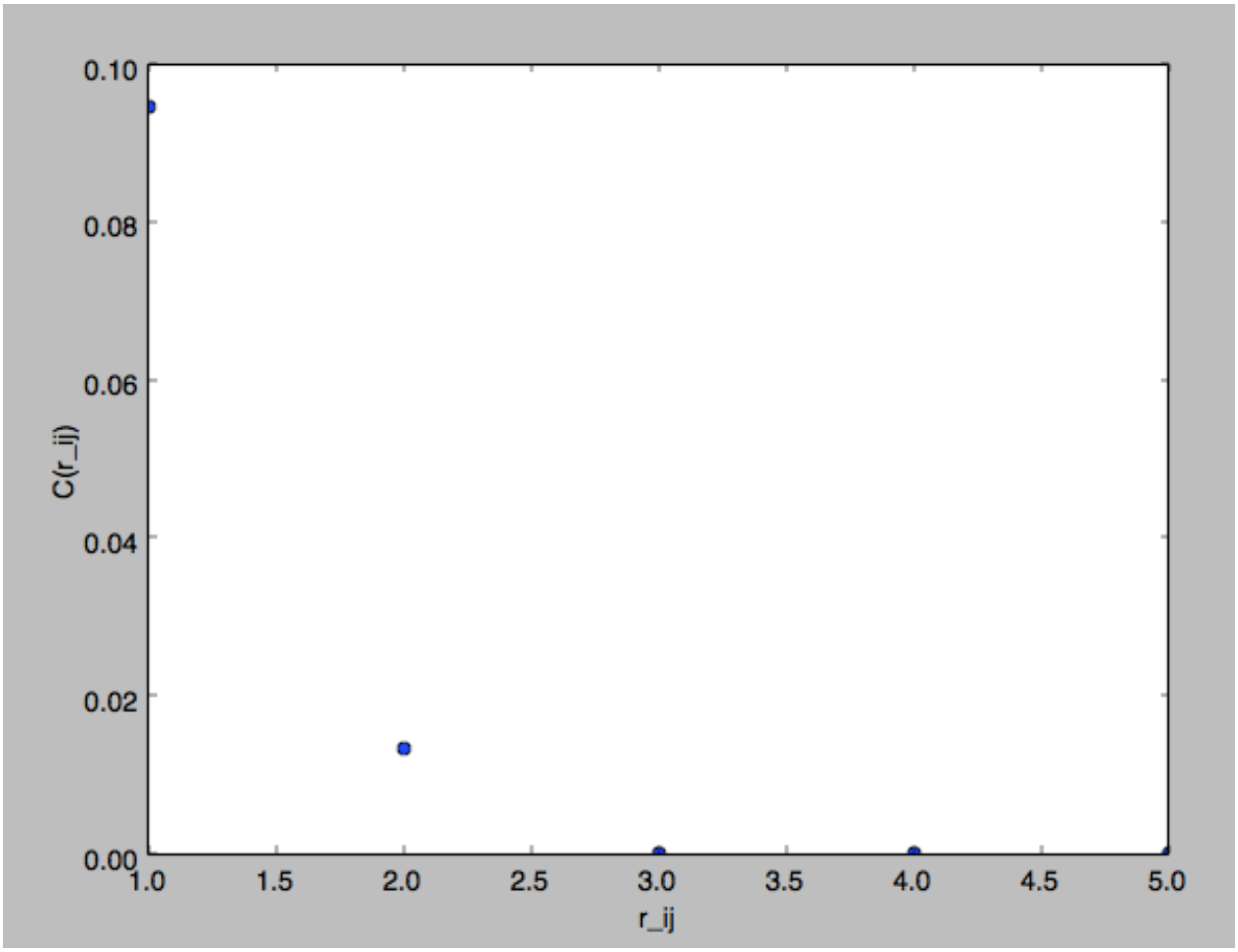
a) See code attached. Note that I modified this code based on my previous homework, which I assumed (perhaps erroneously) to be correct. The main difference is that I now set the coupling constant to be lower (J = 1). I also built in an external field method that took arguments of x and y to respond.

b) Spin correlation plot (T is about half T_C). Look at the nifty interface!:


0
5
10
15
0
5
10
15



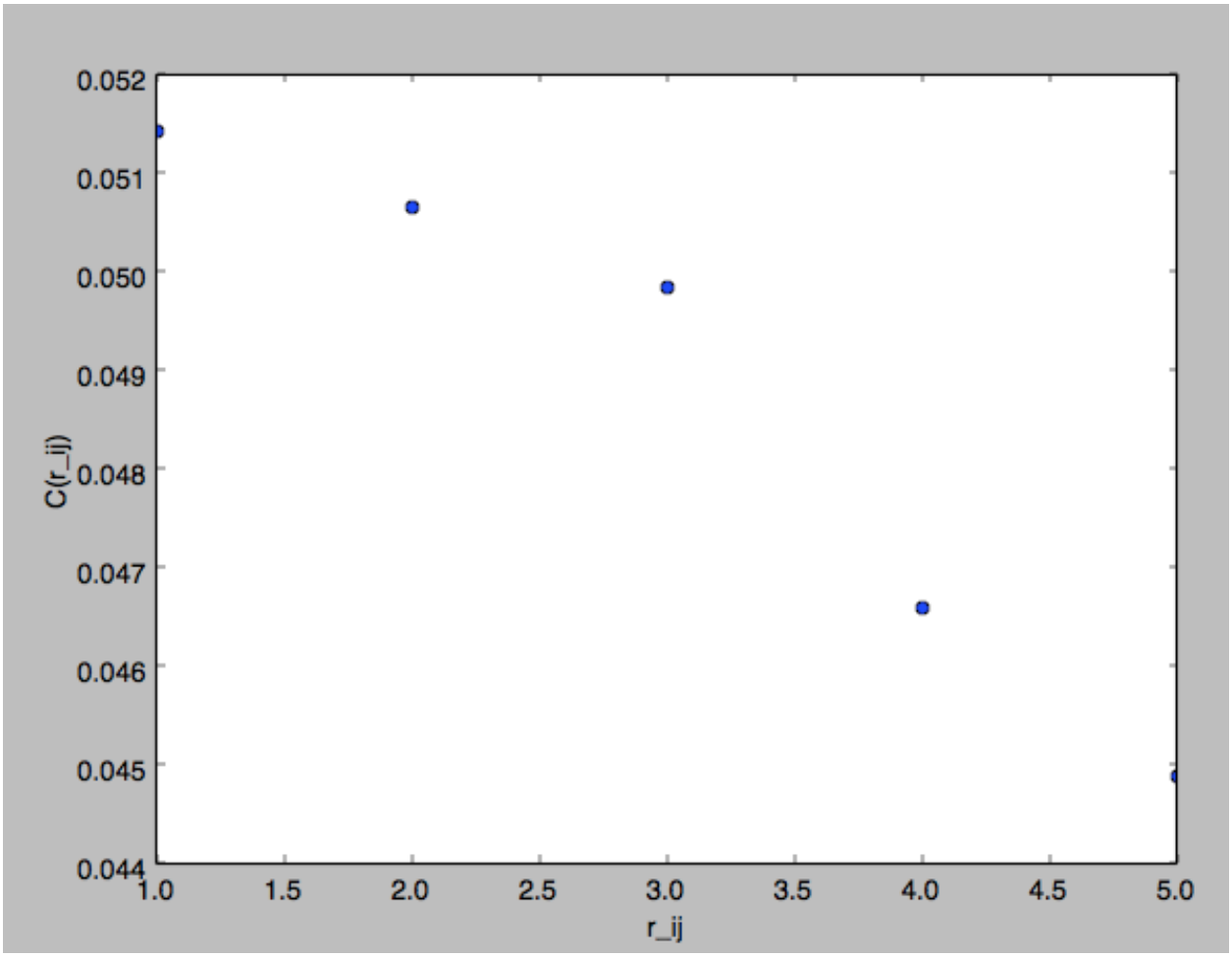
This makes sense, as we are looking at the correlation length along the interface. All the spins are spin up, so we'd expect our function value to be 0 (=<1 * 1> - <1><1>). Looking at the correlation into the bulk:



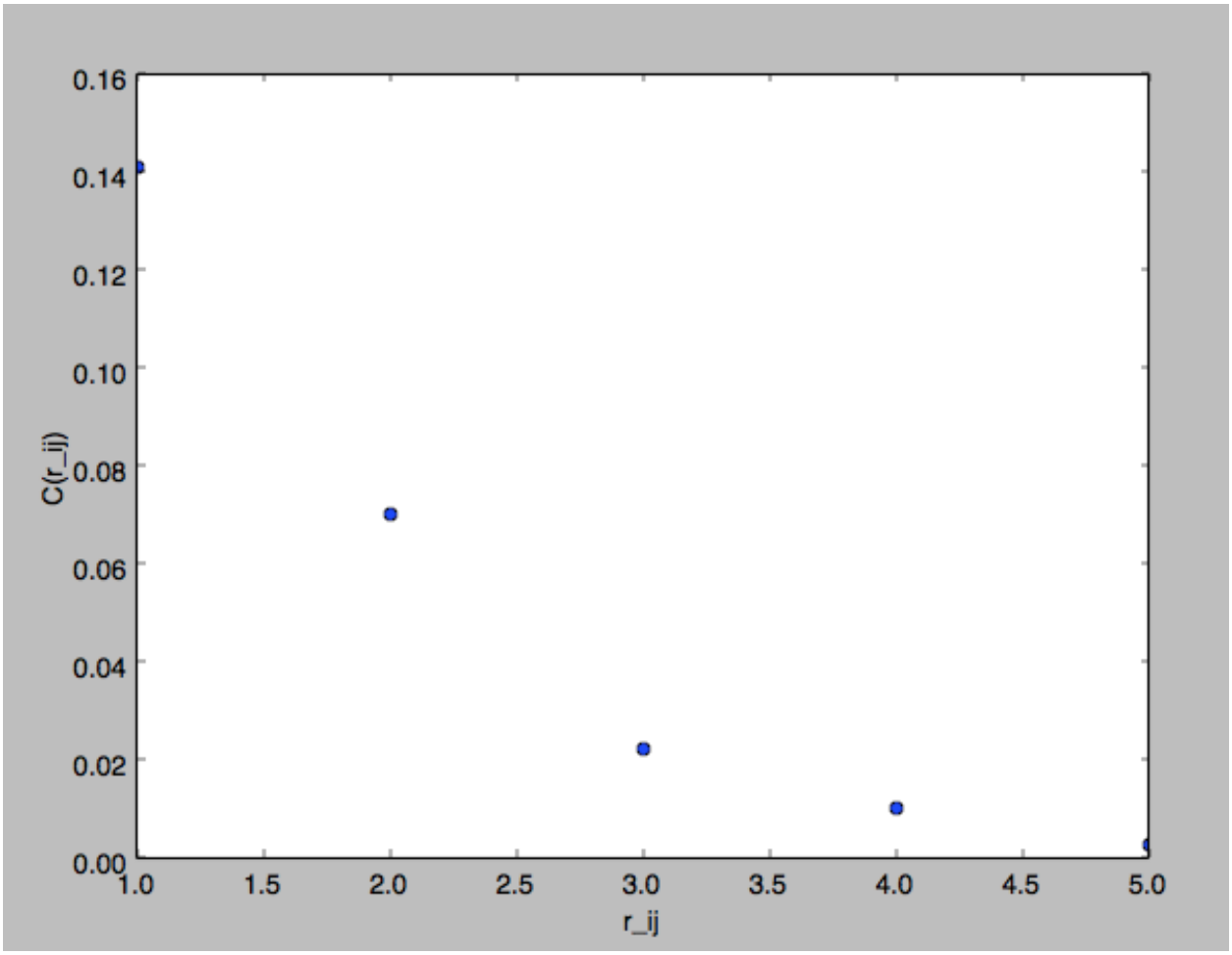
Note the clear drop off as correlation length (this could be sharper, but my run was a little short).

Comments:

c) Spin correlation plot (T is about half T_C) in the middle of the interface (5^{th} position in the layer):



Looking at the correlation length into the bulk:



Comments:

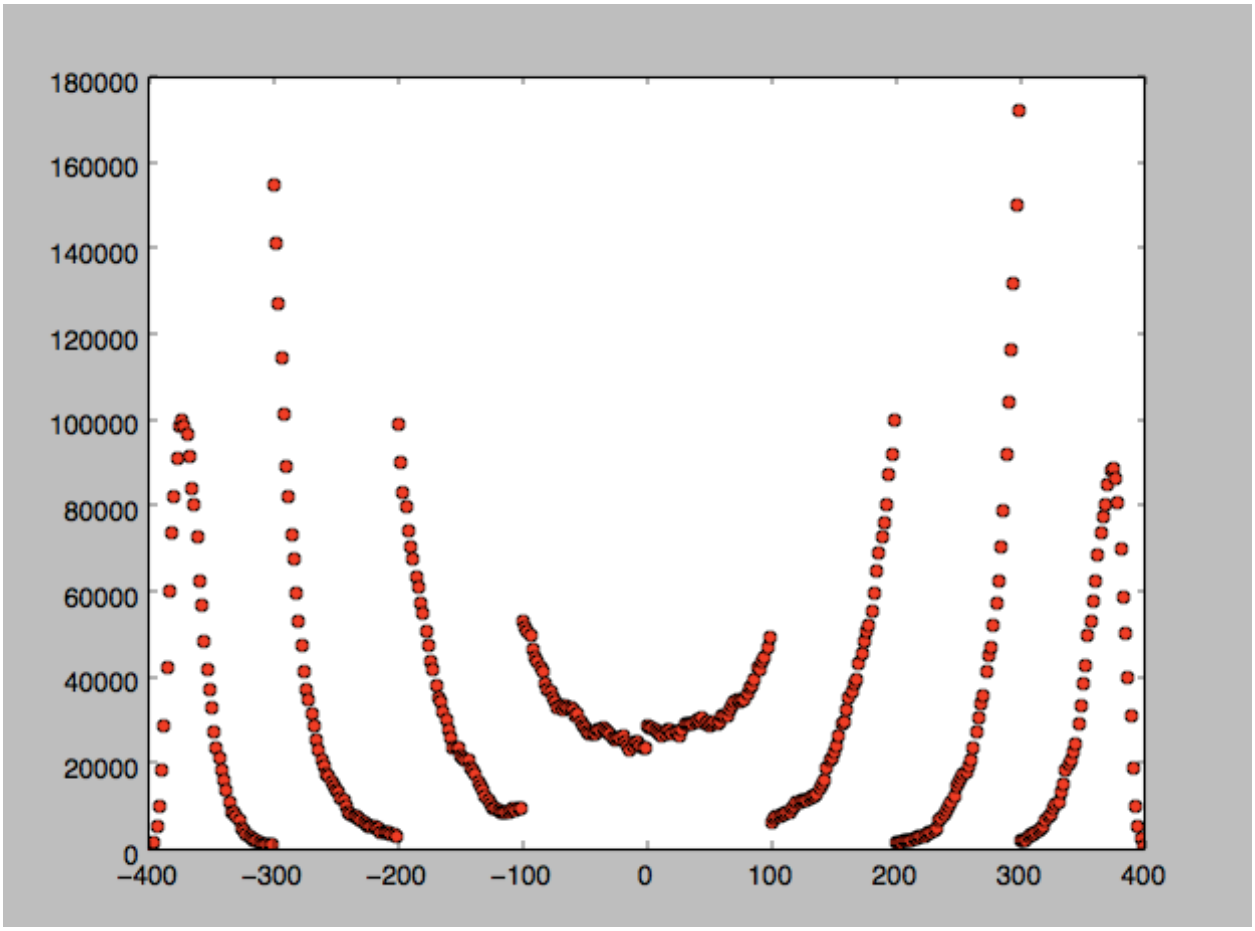
d) In a 40 x 40 system, I'd expect larger bulk solvent effects (if the analogy isn't too sketchy) to take place. The interfaces will penetrate less far into the system.

e) Further comments/expanding: Stabilizing interfaces obviously has many merits. One of the more interesting ones I've read (that vaguely reminds me a lot of this theoretical model) is that of protein/water interfaces. The solvation layer of proteins is incredibly different from that of bulk water. The activity at the interface also has significant implications in the biological activities of the protein, especially as it relates to catalysis.

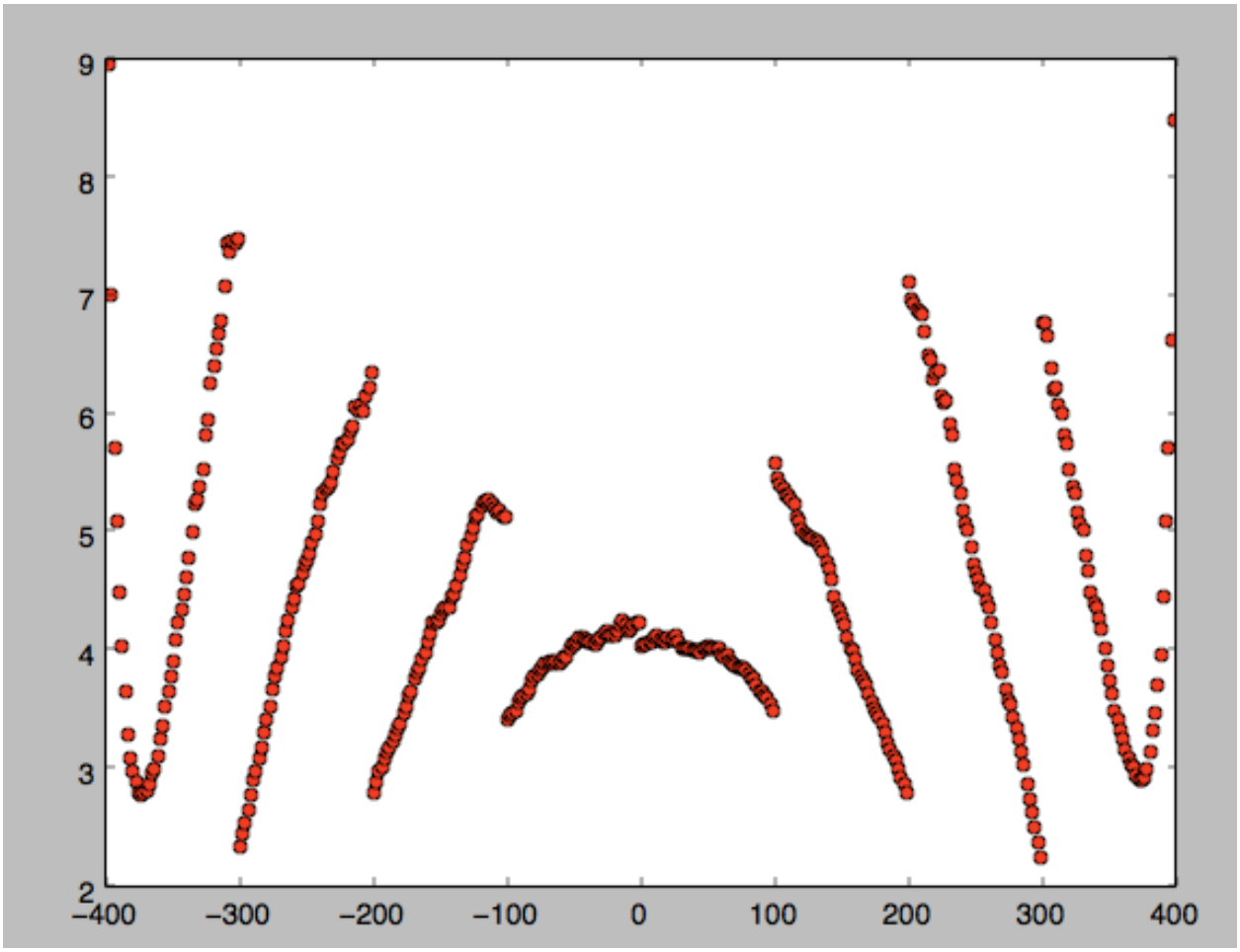
2. Umbrella Sampling Part I

a) Preamble: These are some snapshots of my Umbrella sampling using no magnetic field to prove that I got the thing to work like pg. 174 in Chandler. It has the same $k_BT/J = 2$. I set my coupling constant at about 20 because it seems the deities of computation smiled on my code when I did so. Everything is scaled appropriately given that value for J.

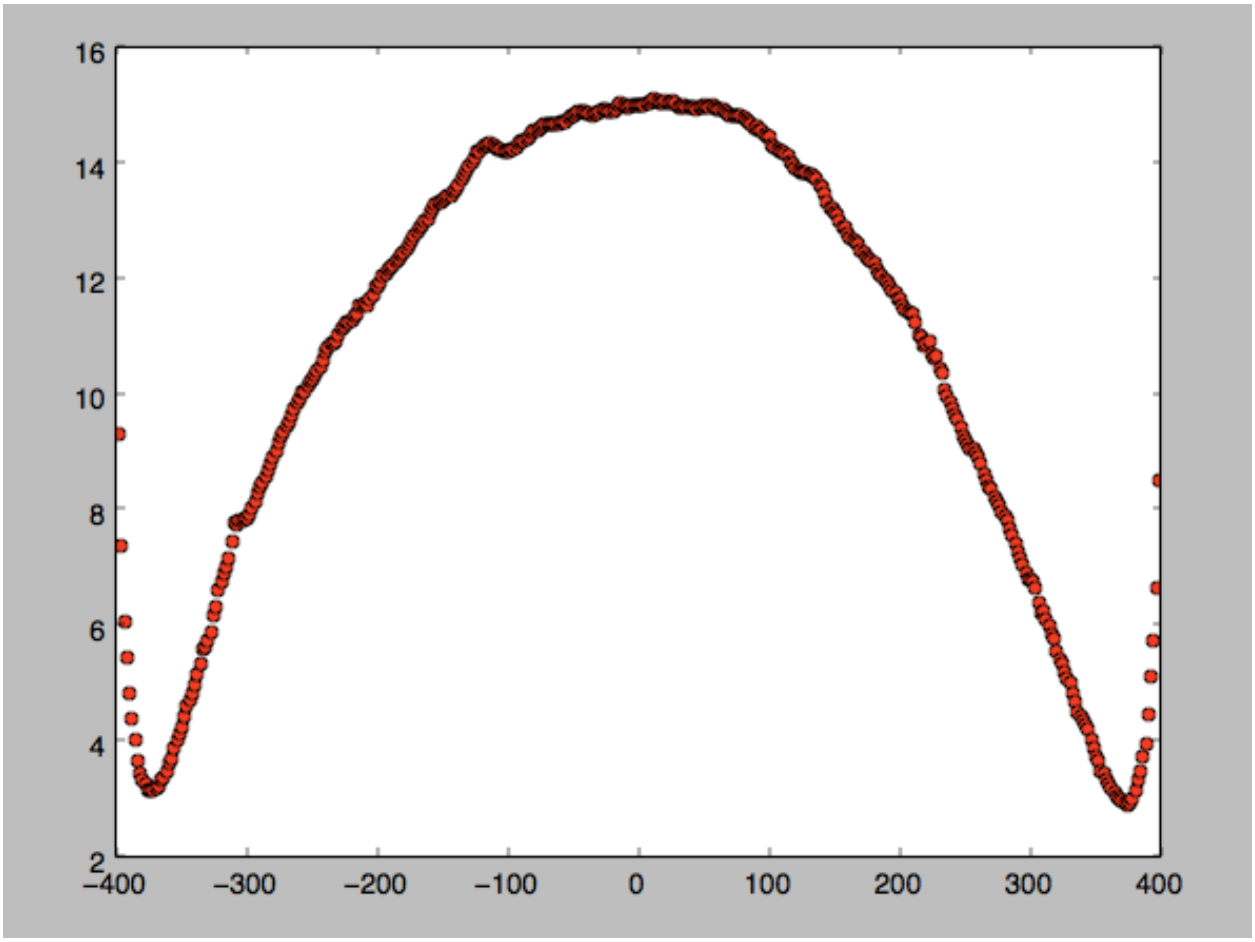
Counts/Umbrella Probability:



-ln P(M):

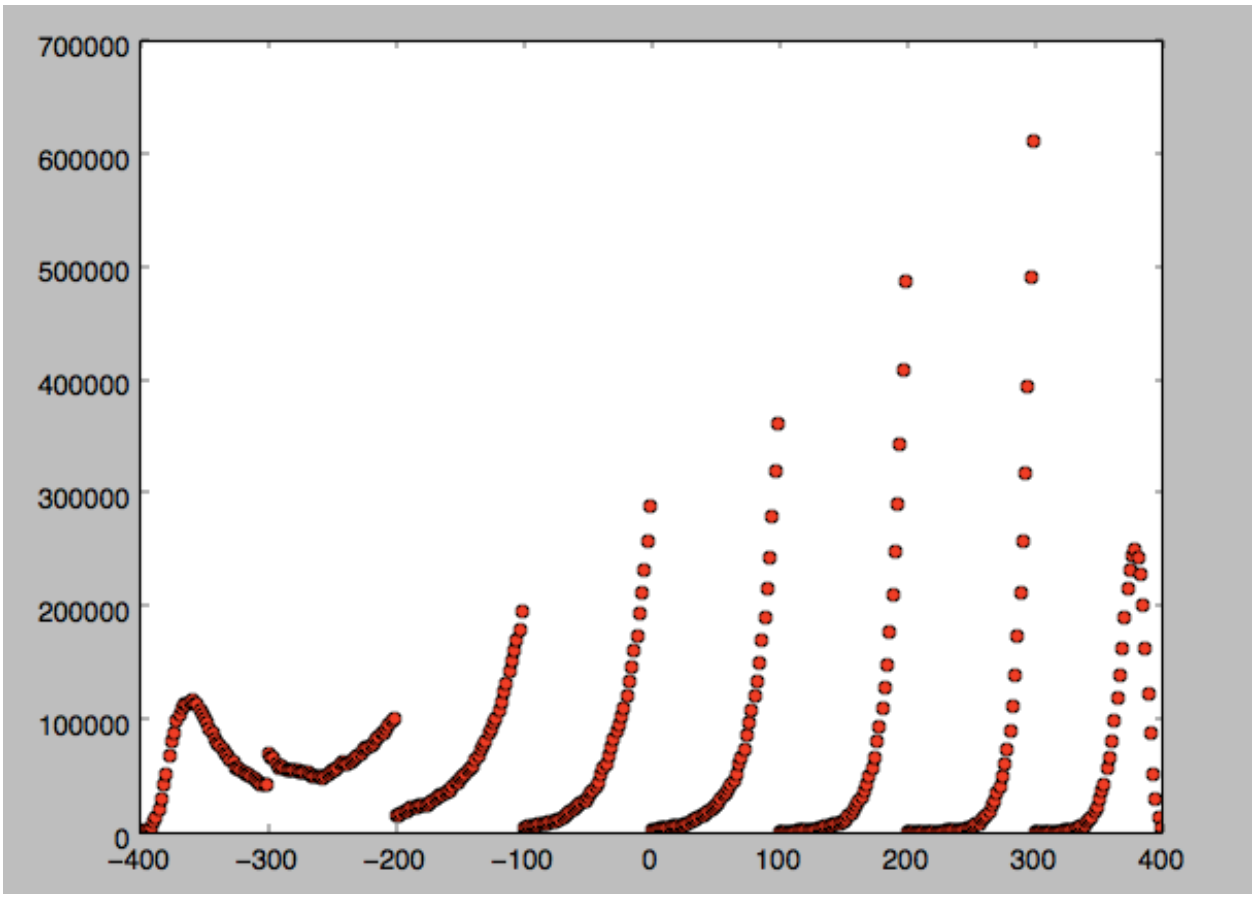


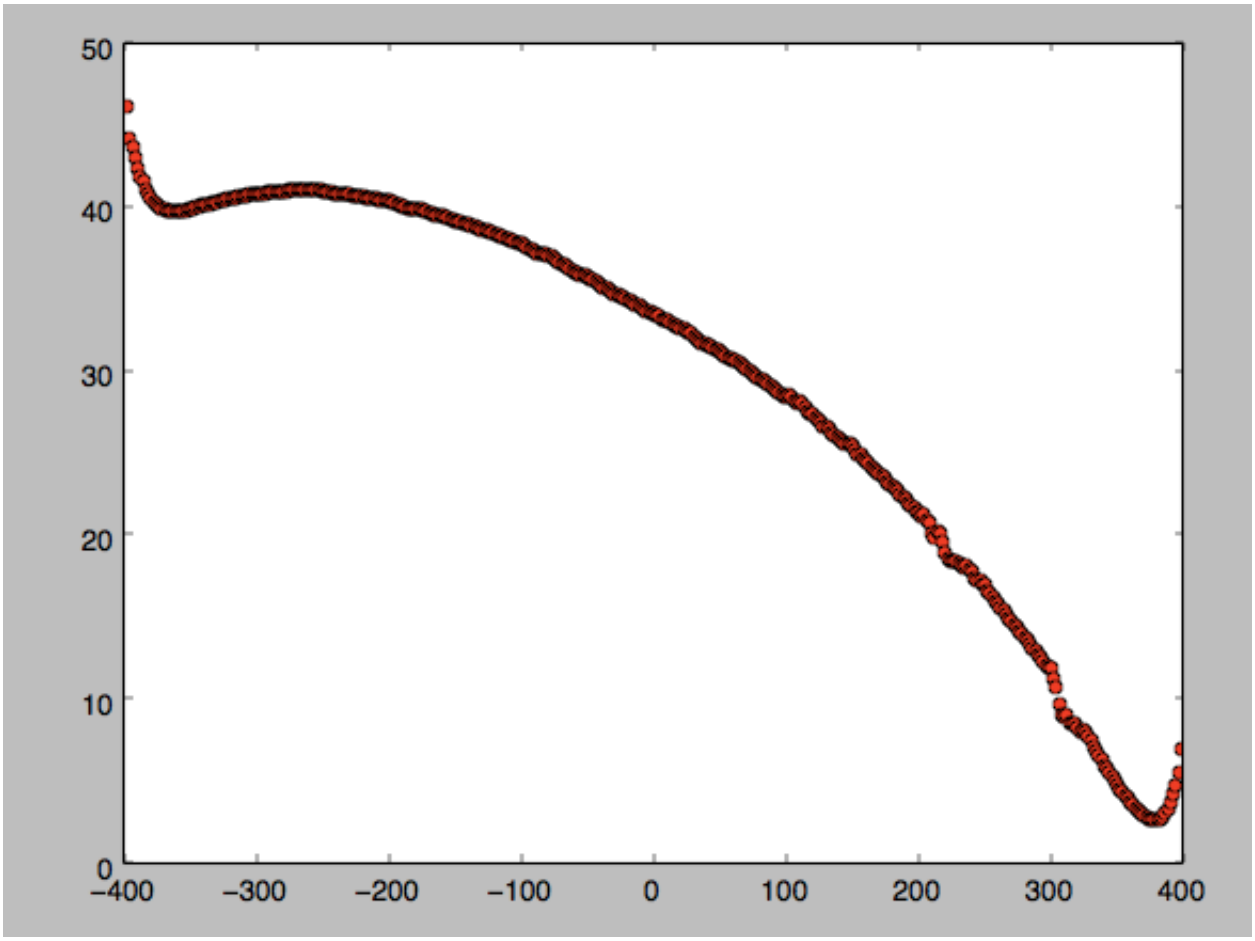
Finally, we have the joined umbrella plot (note that the edge effects I had previously been having problems with are now gone). The units here are Beta * A(M) vs. M:



Now we apply the magnetic field, and show these same plots above/below T_C. These are a slightly lower resolution than the previous example.

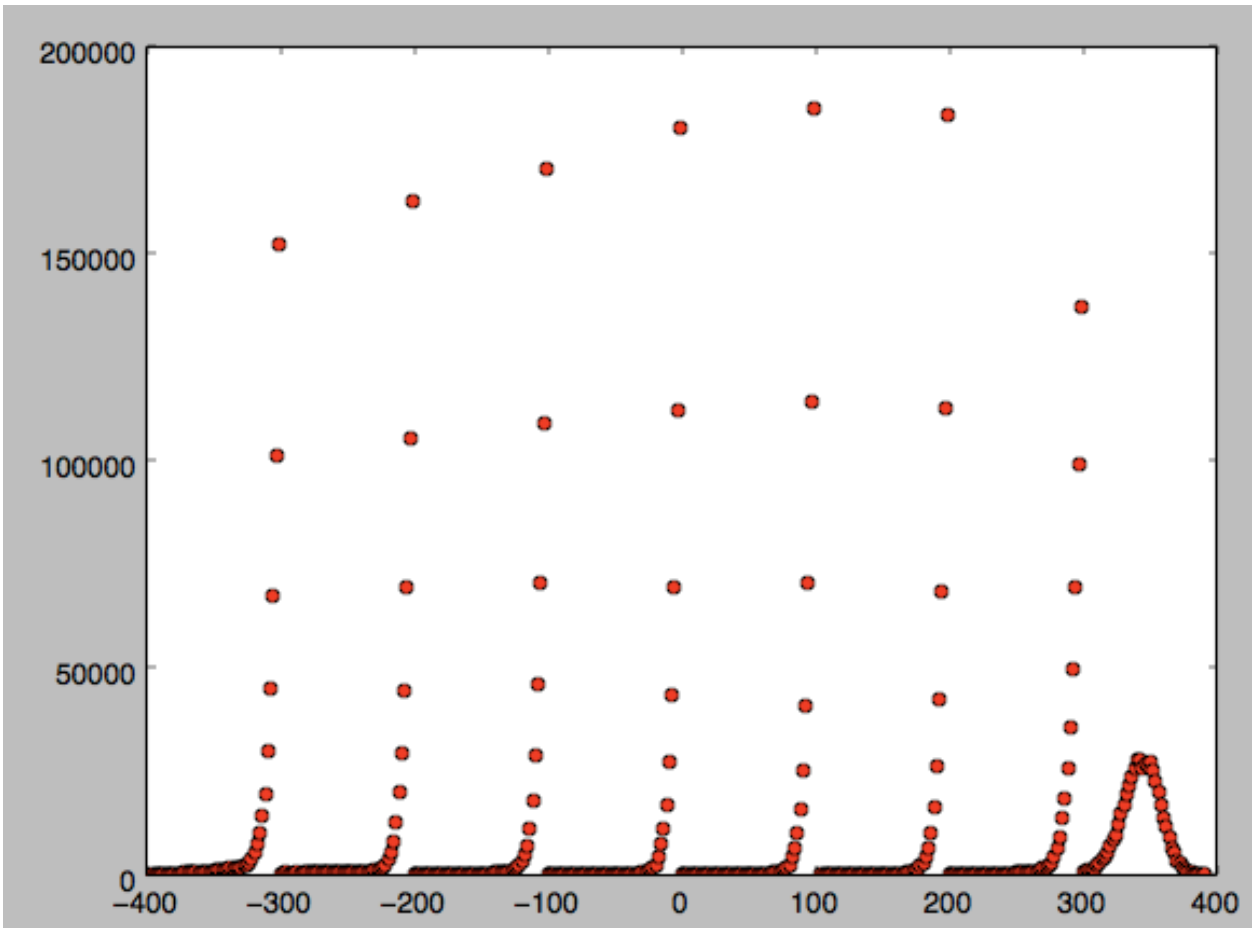
Below T_C ($T/T_C = 2$)

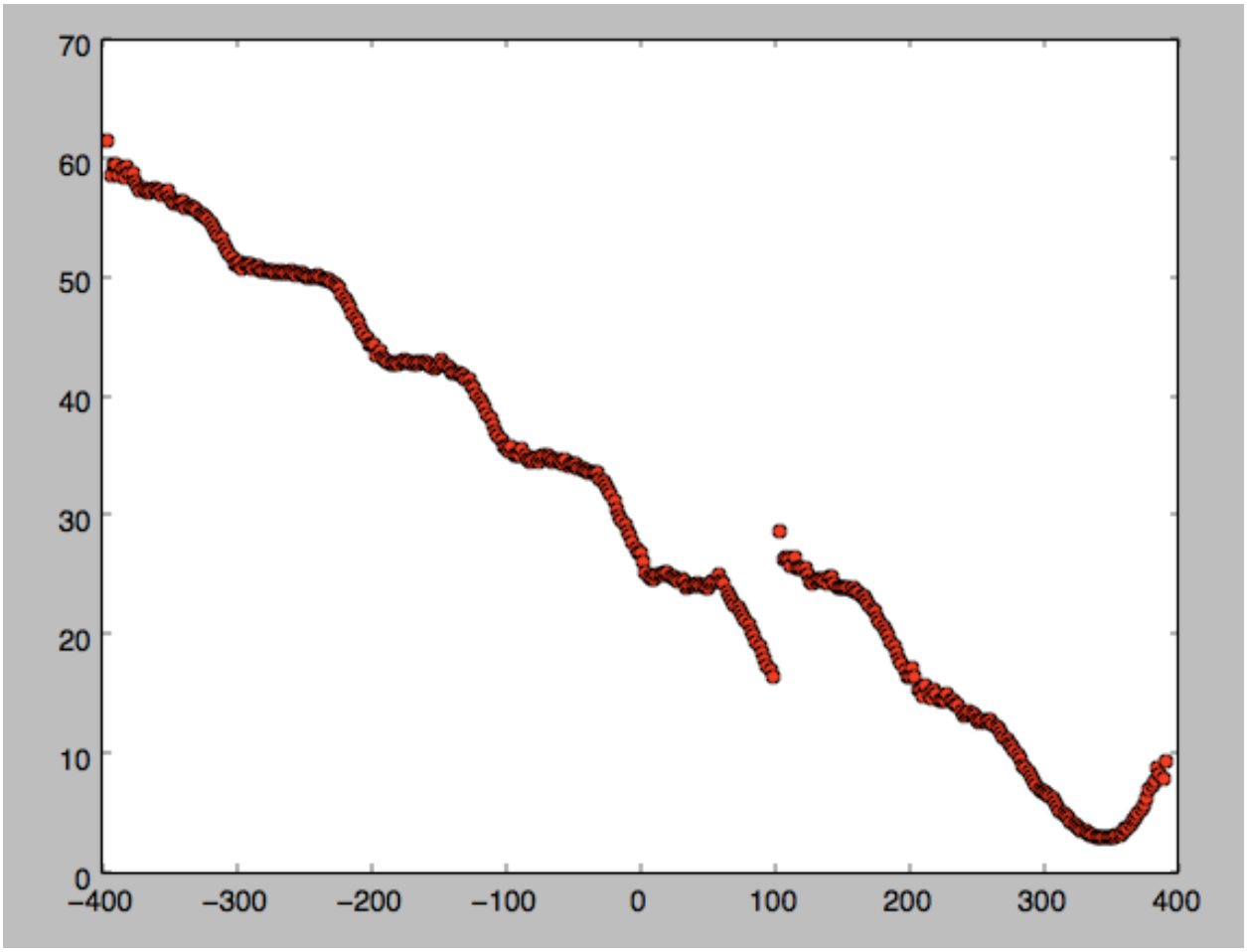

700000
600000
500000
400000
300000
200000
100000
0
−400
−300
−200
−100
0
100
200
300
400



Note how the system is much more comfortable at a positive charge. While local minimum still exist in the negative region, it is far more beneficial for the system to remain at a net positive magnetization (as we would expect).

At $T/T_C = 6$. The slight barrier is now gone and the system just rapidly slides into the positive equilibrium point.


200000
150000
100000
50000
0
−400
−300
−200
−100
0
100
200
300
400



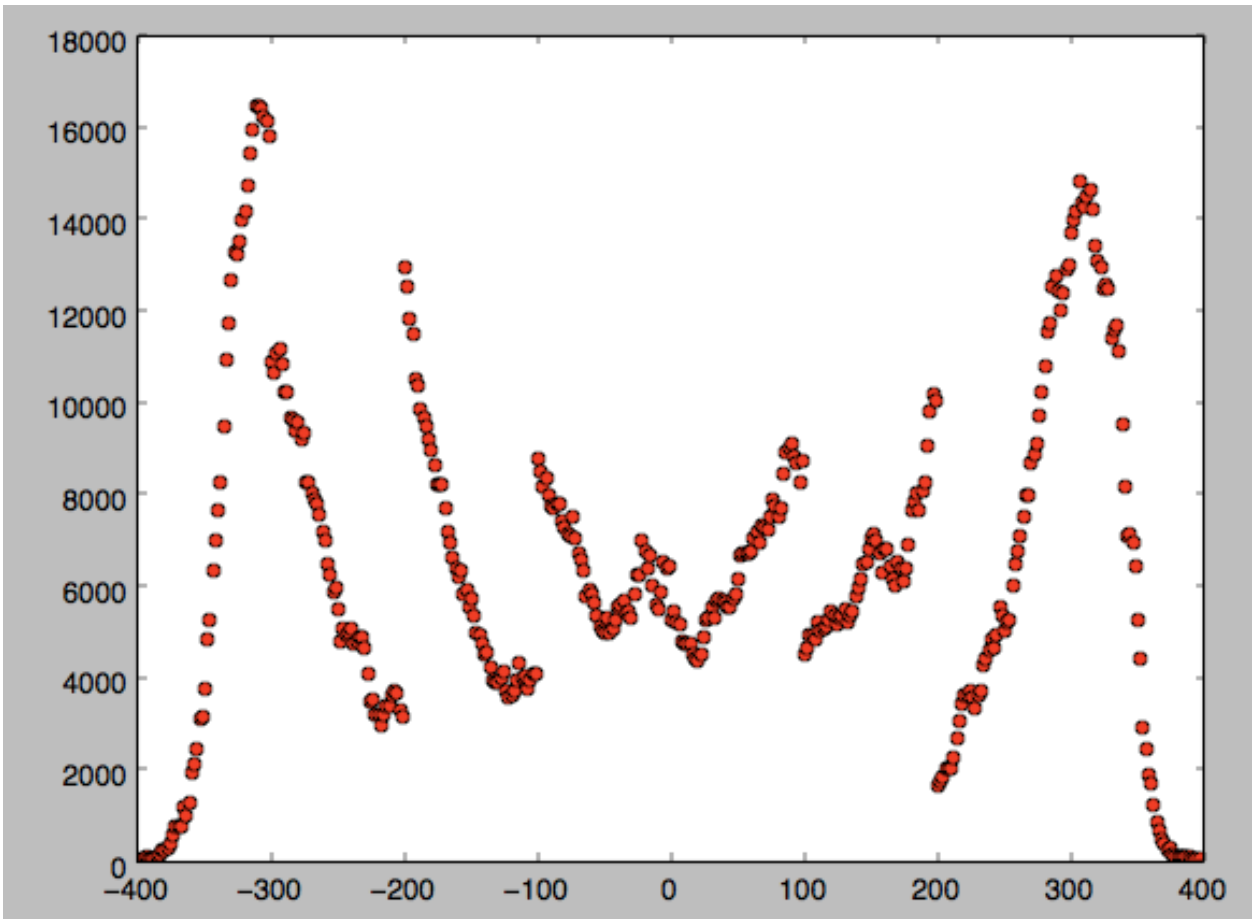
The stitching becomes more difficult in this system, as you can see.

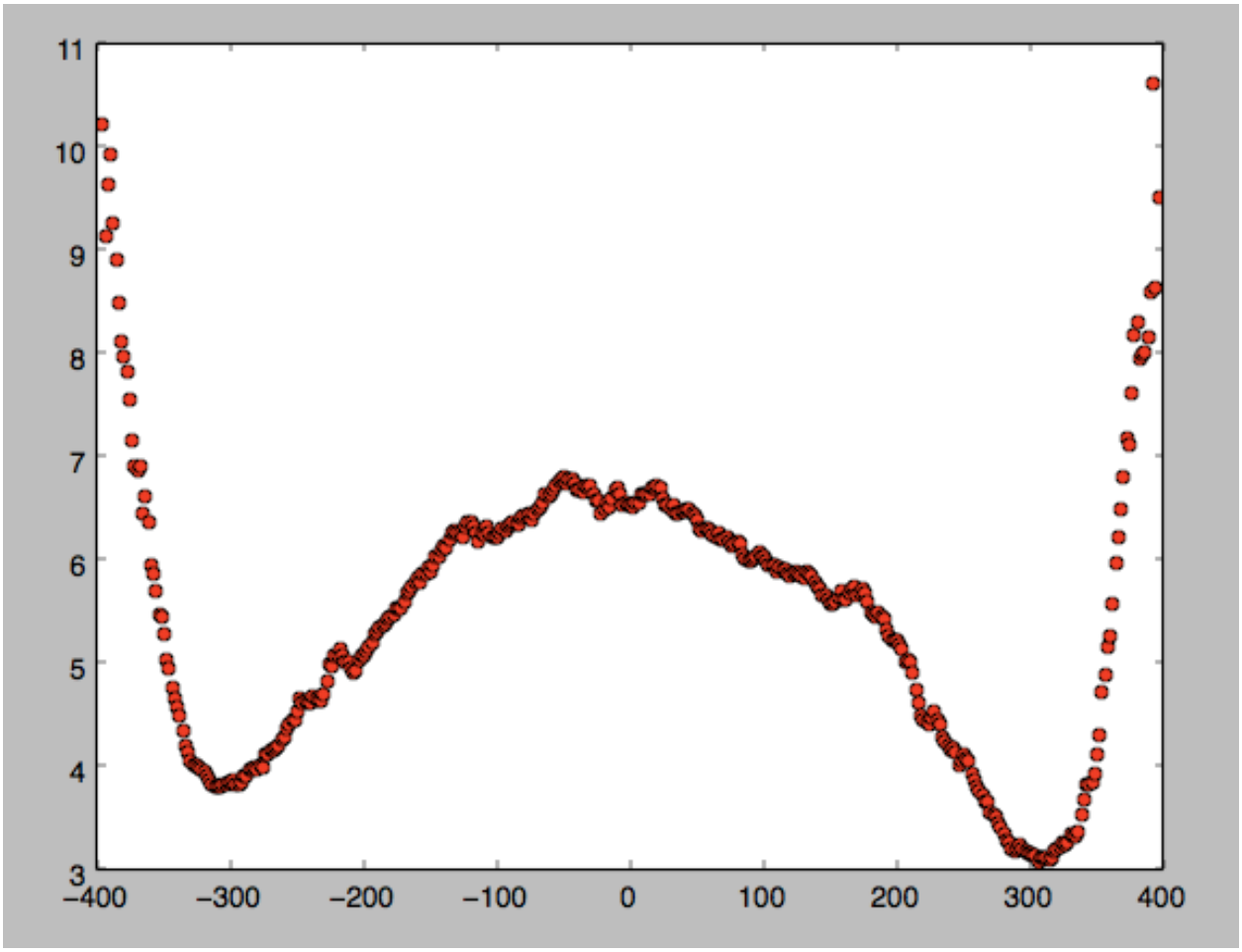
The "big picture" in my opinion of this problem is the visualization of the barrier. Umbrella sampling has most of its utility in studying the system at the full range of trajectories (even the hard to reach ones). This can show us the barrier at normal, perturbed, and perturbed beyond the critical temperature. The graphs give a nice, intuitive sense of what is going on.

I'll save my final comments expanding on umbrella sampling for the end of part III though.

3. Ising Model Part II

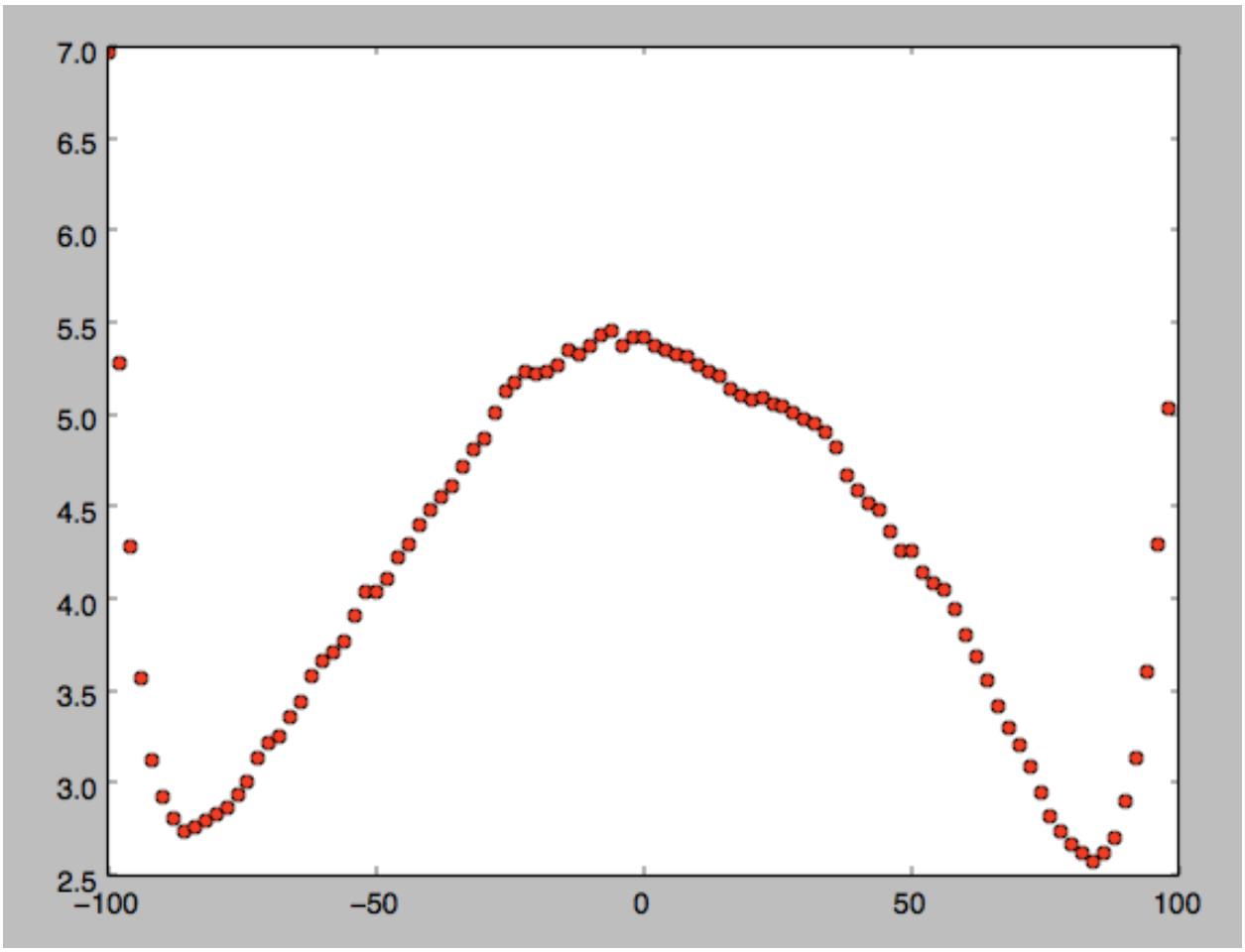
a) Again, my simulation time might be a little short but I think you get the idea for the first one what it can look like. My computer is really bad, and generating these things can take a while.:


18000
16000
14000
12000
10000
8000
6000
4000
2000
0
−400
−300
−200
−100
0
100
200
300
400

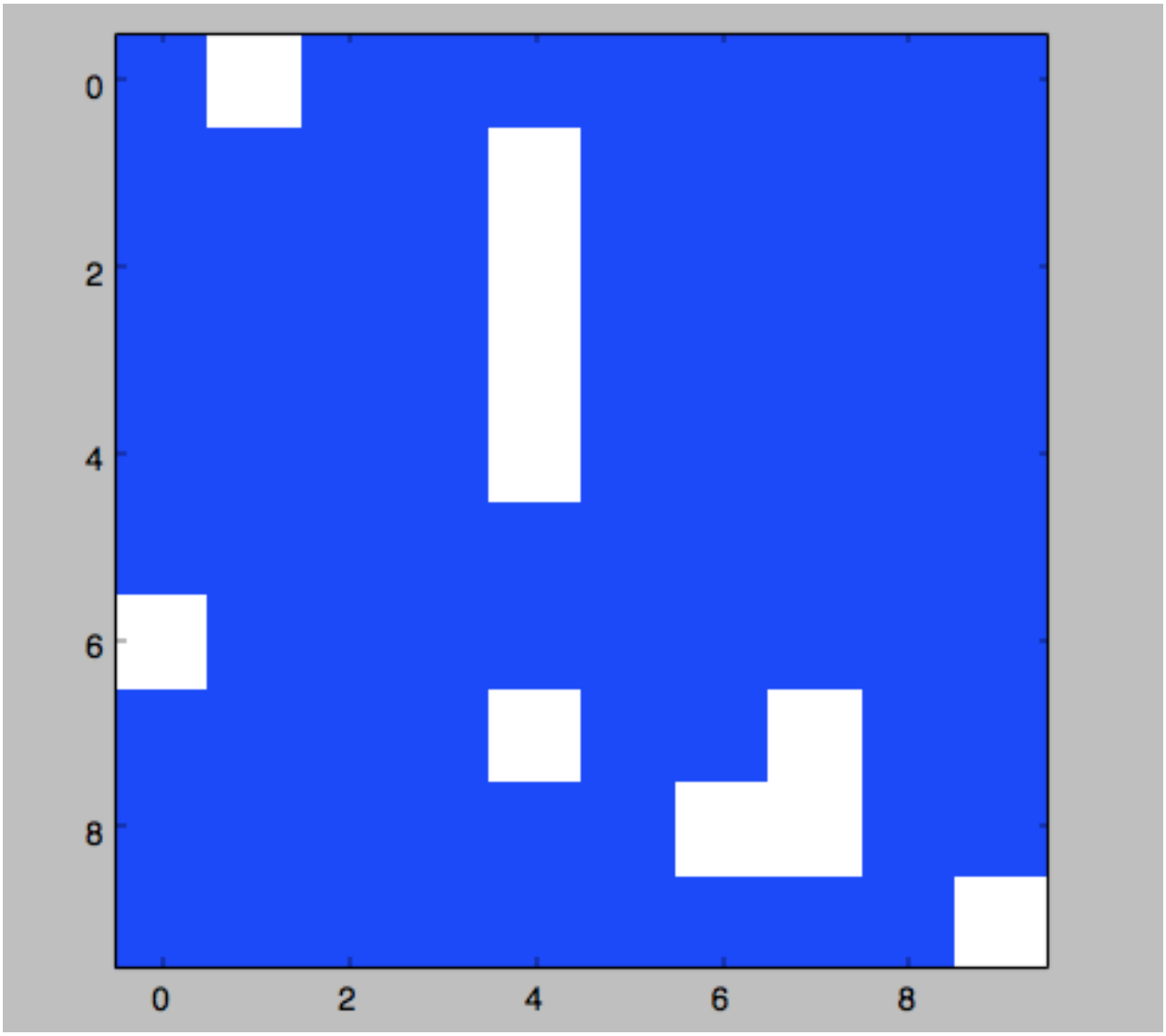


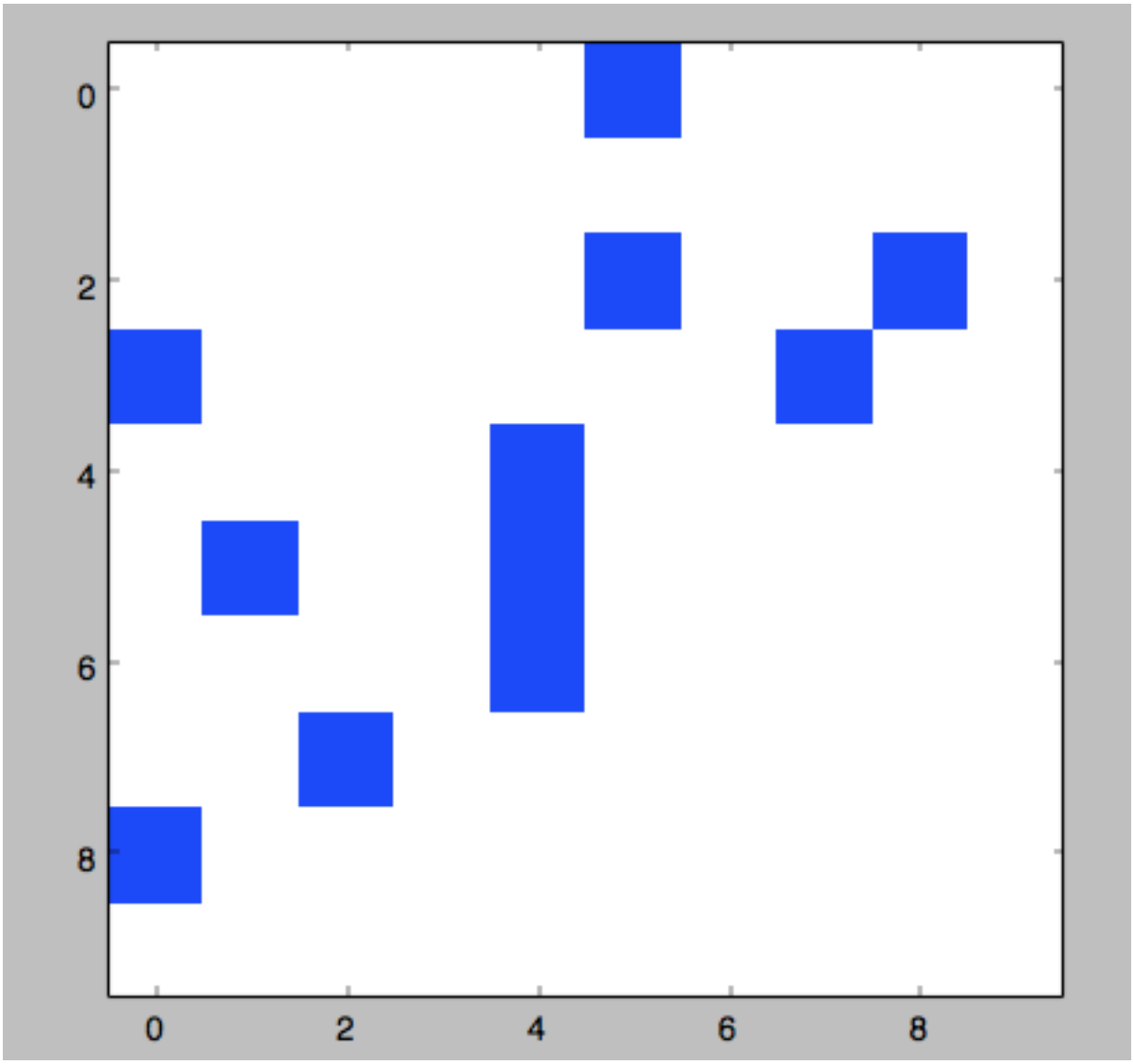
Again, our y-axis is –Beta Ln(P(M)). Moving from one state to the other simply requires us to move from one equilibrium point to the other… here we have Beta Delta A = Beta A(0) – Beta A(~300) = Beta (6.5 – 3.1) = Beta 3.5. Notice how our once high peak has flattened! This makes sense as it is easier for the system to cross over the barrier at a higher temperature where symmetry can be broken more easily.

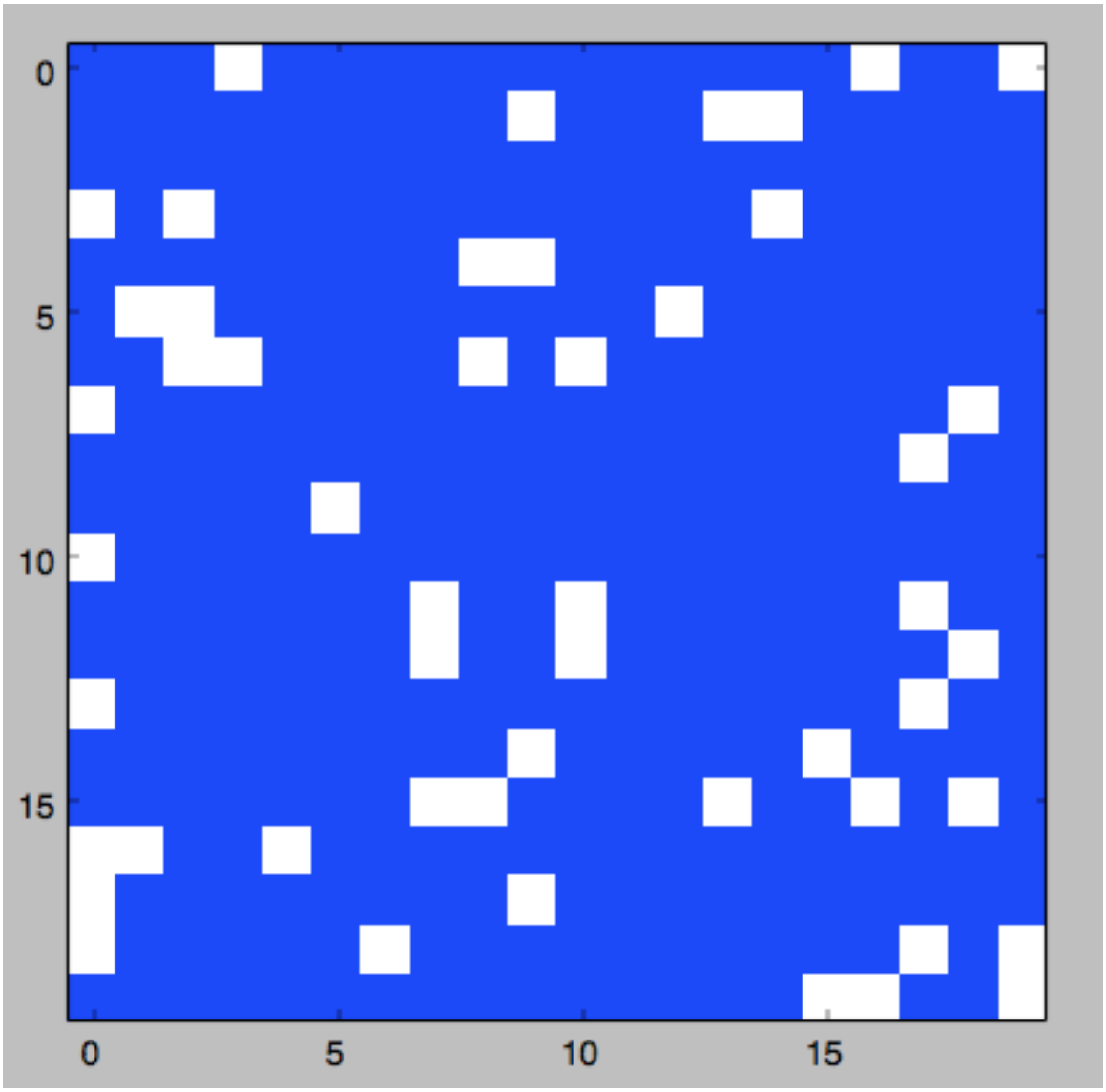
I fiddled with the 10 x 10 a bit. It only has 4 regions and some different edge buffers to get it to work out correctly.

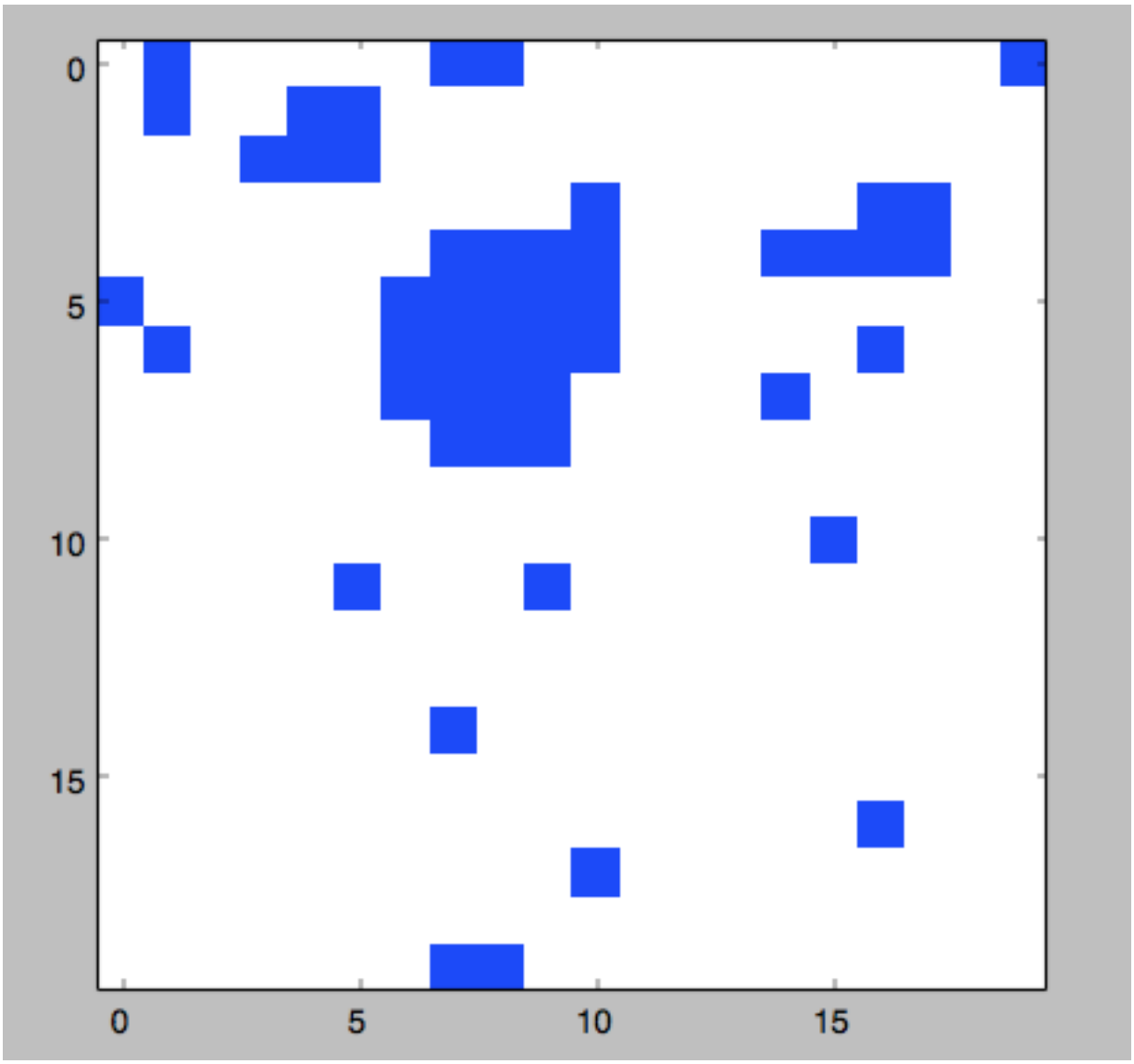


Here the difference is Beta Delta A = Beta (5.4 – 2.6) = Beta (2.4). As expected, size matters, since energy is an extensive property. Some representative plots for the minimum values (blue is negative):


0
2
4
6
8
0
2
4
6
8


0
2
4
6
8
0
2
4
6
8


0
5
10
15
0
5
10
15



Assuming you cranked the temperature on the system up very high, the barrier would disappear entirely.

ii) For the determination of magnetic field, we take the double derivative of our Beta A(M) plot at equilibrium. The value doesn't really (and ignoring numerical artifacts, doesn't at all) depend on system size, as it is an intensive variable.

Given that I don't know numpy well enough to build an automated second derivative filter with correct smoothing, I'm going to build a table of min points and do a manual second derivative about the positive equilibrium value with one of my GSI's recommendation of 5 points.

Applied Field (C = 20)	Beta A(M = equilibrium)
-2	-49.03
-1	-45.94
0	2.91
1	2.85
2	2.82

Applied Field (C = 20)	Beta A'(M = equilibrium)
-2	N/A
-1	3.09
0	48.85
1	-0.14
2	-0.04

Applied Field (C = 20)	Beta A''(M = equilibrium)
-2	N/A
-1	N/A
0	45.76
1	-48.99
2	-0.18

So the magnetic susceptibility is 45.76. This could be massively thrown off through numerical error, but hopefully the methodology was presented sufficiently.

At 10x10:

Applied Field (C = 20)	Beta A(M = equilibrium)
-2	-12.35
-1	-11.03
0	1.74
1	1.09
2	1.08

Which seems consistent.

Final thoughts: This semester I've been trying to think about statistical mechanics as it relates to biophysical modeling. An area that has always interested me is the visualization of rare events. Despite their infrequence, many of the most interesting cellular events happen at so sporadically that sophisticated computational techniques are needed to actually visualize them. Umbrella sampling was a great introduction into the underlying principles. For a calculation of a rate constant using

transition state theory (TST), for example, one often needs to umbrella sampling as part of a suite of tools.

```
import numpy
import random
import math
import matplotlib.pyplot as plt

CC = 1 # Coupling constant
ExField = 0.0
H = 3
#This method is essentially a collection of all of the steps of the Monte Carlo
process
#Its arguments are:
#n_steps: The number of steps you'd like the monte carlo algorithm to iterate
#edge_length: The edge dimension of your lattice.  So, for a 20x20 lattice, this would
equal 20
#ext_field: The strength of the external field
#temp: The temperature
#start_flag: An optional flag which, if set to 1 will read a starting configuration
from an external file.
#config_file: Another optional flag which specifies the external file which
initializes the lattice.

def ExternalFieldBias(x,y):
    if y < 1:
        if x < 10:
            return -1 * H
        else:
            return H
    else:
        return 0

def PerformRun(n_steps,edge_length,ext_field,temp,start_flag=0,config_file=None):
    LatticeEdgeDimension = edge_length
    ExternalField = ext_field
    T = temp

    SpinLattice, energy, netmag =
initialize(start_flag,LatticeEdgeDimension,ExternalField,config_file)
    SpinLattice, energy, netmag, stderr, cij =
monte_carlo(n_steps,SpinLattice,T,energy,netmag)
    write_config("FinalConfig.out",SpinLattice)
    return SpinLattice, energy, netmag, stderr, cij

#Initialize, if given a start_flag of 0, initializes the lattice to be entirely
magnetized.
#If start_flag is 1, it reads from config_file for the lattice configuration
def initialize(start_flag,edge_length,field,config_file=None):
    SpinLattice = [[0 for x in xrange(edge_length)] for y in xrange(edge_length)]
    netmag = 0
    energy = 0.
    ExField = field

    if (start_flag==0):
        print "Generating magnetized configuration...\n"
        for x in xrange(edge_length):
            for y in xrange(edge_length):
                SpinLattice[x][y]=1
                netmag += 1

        energy = -2.0*(edge_length**2)
```

```
    else:
        print "Reading configuration from file: " + str(config_file)

        SpinLattice = read_config(config_file)
        edge_length = len(SpinLattice[0])
        for x in xrange(edge_length):
            for y in xrange(edge_length):
                netmag += SpinLattice[x][y]
                up = 0
                right = 0

                if (x==edge_length-1):
                    right=SpinLattice[0][y]
                else:
                    right=SpinLattice[x+1][y]

                if (y==edge_length-1):
                    up=SpinLattice[x][0]
                else:
                    up=SpinLattice[x][y+1]

                energy -= SpinLattice[x][y]*(up+right)

    energy -= field*netmag

    print "Initial magnetization per spin = " + str(netmag/(edge_length**2)) + "\n"
    print "Initial energy per spin = " + str(energy/(edge_length**2)) + "\n"

    return SpinLattice,energy,netmag

#This method contains the bulk of the calculation.  Code that collects statistics,
however, is missing.
def monte_carlo(n_steps,spin,T,energy,netmag):

    avmag=0.0
    aven=0.0
    edge_length = len(spin[0])

    HOWFAR = 6

    avmagList = []
    xlen = edge_length
    ylen = edge_length
    cij = [-1 for i in range(HOWFAR)] # Assume xlen = ylen...
    productRij = [[] for i in range(HOWFAR)]
    jSpinRij = [[] for i in range(HOWFAR)]
    mySpinList = []

    with open("trajectory.dat",'w') as f:

        #Each 'step' of the algorithm attempts edge_length**2 trial moves.
        print "Generating trajectory...\n"
        for step in xrange(n_steps):
            for j in xrange(edge_length**2):
                x = int(edge_length*numpy.random.rand())
                y = int(edge_length*numpy.random.rand())

                spin,energy,netmag = trial_move(x,y,spin,energy,netmag,T)

                xc, yc = 9, 0
```

```
                mySpin = float(spin[xc][yc])
                mySpinList.append(mySpin)
                for delta_x in range(HOWFAR): #TODO: Fix hard coding
                    '''Sj1 = float(spin[(xc + delta_x) % 20][yc])
                    # Sj2 = float(spin[(xc - delta_x) % 20][yc])
                    productRij[delta_x].append(mySpin * Sj1)
                    jSpinRij[delta_x].append(Sj1)
                    # productRij[delta_x].append(mySpin * Sj2)
                    # jSpinRij[delta_x].append(Sj2)'''


                    # Code for going "into" bulk
                    Sj1 = float(spin[(xc)][(yc + delta_x) % 20])
                    # Sj2 = float(spin[(xc - delta_x) % 20][yc])
                    productRij[delta_x].append(mySpin * Sj1)
                    jSpinRij[delta_x].append(Sj1)
                    # productRij[delta_x].append(mySpin * Sj2)
                    # jSpinRij[delta_x].append(Sj2)

            avmag = (avmag * (step) + netmag) / (step + 1) # step = (number of moves)
- 1
            aven  = (aven  * (step) + energy) / (step + 1)

            #if (step%10==0):
            #    f.write("\t".join([str(s) for s in [step,netmag/
(edge_length**2),energy/(edge_length**2)]]) + "\n")


        for r in range(HOWFAR):
            print r
            print numpy.mean(productRij[r])
            print float(sum(productRij[r]))/len(productRij[r])
            print numpy.mean(jSpinRij[r])
            print numpy.mean(mySpinList)
            cij[r] = (float(sum(productRij[r]))/len(productRij[r]) -
(float(sum(jSpinRij[r]))/len(jSpinRij[r]) * float(sum(mySpinList))/len(mySpinList)))
            # raw_input()

        #/* Output averages */
        print "Average magnetization per spin = " + str(avmag/
(n_steps*edge_length**2)) + "\n"
        print "Average energy per spin = " + str(aven/(n_steps*edge_length**2)) + "\n"
    stderr = (numpy.std(avmagList)/(n_steps**.5))
    return spin, energy, netmag, stderr, cij

#This method actually attempts the trial move, and decides whether to accept or reject
the trial.
def trial_move(x,y,spin,energy,netmag,T):
    neighbor_mag=0
    edge_length = len(spin[0])
    up,down,left,right = 0,0,0,0
    deltae = 0.0

    if (x==0):
        left=spin[edge_length-1][y]
    else:
        left=spin[x-1][y]
    if (x==edge_length-1):
        right=spin[0][y]
    else:
```

```
        right=spin[x+1][y]
    if (y==edge_length-1):
        up=spin[x][0]
    else:
        up=spin[x][y+1]
    if (y==0):
        down=spin[x][edge_length-1]
    else:
        down=spin[x][y-1]

    #/* left, right, up, and down are the states */
    #/* of spins neighboring (x,y) */
    #/* compute change in energy if spin[x][y] were flipped, */
    my_old_spin = spin[x][y]
    neighbor_sum = up + down + left + right
    delta_E = 2 * ExternalFieldBias(x,y) * my_old_spin + 2 * CC * my_old_spin *
neighbor_sum

    #/*************************************************************/

    #/* accept according to Metropolis Monte Carlo rules */
    #/* update magnetization and energy if necessary */
    if (min(1, math.exp(-1 * delta_E / T)) > random.random() ):
        # Flip the spin if min(1, e^(-beta E)) > random float (from [0, 1))
        energy += delta_E
        spin[x][y] = -1 * my_old_spin
        netmag -= (2 * my_old_spin)
    #/*************************************************************/

    return spin, energy, netmag


#Writes a configuration to file.  Spins are written as integers, tab delimited.
def write_config(filnam,spin):
    with open(filnam,'w') as f:
        for x in xrange(len(spin[0])):
            f.write(str("\t".join([str(s) for s in spin[x]])) + "\n")

#Reads from a configuration file.  Reads from the same format as the write_config
method above--i.e. tab delimited integers, with an equal
#number of rows as columns.
def read_config(filnam):
    spin = []
    with open(filnam,'r') as f:
        for line in f:
            ThisRow = []
            for s in line.split("\t"):
                ThisRow.append(int(s))
            spin.append(ThisRow)

    return spin

def createSpinPlot(SpinLattice):
    edge_length = len(SpinLattice)
    SpinColor = [[0 for x in xrange(edge_length)] for y in xrange(edge_length)]
    for y in range(edge_length):
        for x in range(edge_length):
            if SpinLattice[x][y] == 1:
                SpinColor[x][y] = [1.0, 1.0, 1.0]
            else:
```

```
                SpinColor[x][y] = [0.0, 0.0, 1.0]
    SpinArray = numpy.array(SpinColor)
    plt.imshow(SpinArray, interpolation='nearest')
    plt.show()
    plt.clf()

#This is how you might call the program to perform 10000 steps in a 20x20 lattice at 0
applied field and at 1.0 K, where the lattice is initially fully magnetized

rPoints, cPoints = [], []
tempPoints = []
amagPoints = []
initialTemp = 1.0
step = 1

for T in numpy.linspace(0,9,100):
    print T
    SpinLattice, energy, netmag, stderr, cij = PerformRun(100,20,0.0,T)
    tempPoints.append(T)
    amagPoints.append(netmag)
    # if deltaT == 13 or deltaT == 49 or deltaT == 99:
    #    createSpinPlot(SpinLattice)
    #    createCriticalPlots(SpinLattice)

    if T == 1.0 or T == 2.0:
        createSpinPlot(SpinLattice)
        for r in range(1,6):
            rPoints.append(r)
            cPoints.append(cij[r])
        plt.plot(rPoints,cPoints,'bo')
        plt.xlabel("r_ij")
        plt.ylabel("C(r_ij)")
        plt.show()
        plt.clf()
        rPoints = []
        cPoints = []

plt.plot(tempPoints,amagPoints,'ro')
plt.show()

#This is how you might call a program to perform the same, except on a user-specified
input file:
#Note that when using a user-specified input file, the edge dimension argument is
ignored (in this case, 20 is ignored, and the
#edge dimension is set to the dimensions of the input file lattice
#PerformRun(10000,20,0.0,1.0,1,"startconfig.in")
```

```
import numpy
import random
import math
import matplotlib.pyplot as plt
import time

CC = 20 # Coupling constant
ExField = 0
UmbrellaWindows = 8
EdgeBuffer = 1

#This method is essentially a collection of all of the steps of the Monte Carlo
process
#Its arguments are:
#n_steps: The number of steps you'd like the monte carlo algorithm to iterate
#edge_length: The edge dimension of your lattice.  So, for a 20x20 lattice, this would
equal 20
#ext_field: The strength of the external field
#temp: The temperature
#start_flag: An optional flag which, if set to 1 will read a starting configuration
from an external file.
#config_file: Another optional flag which specifies the external file which
initializes the lattice.

def combineHistograms(new, total):
    for k in new.keys():
        if k in total.keys(): total[k] += new[k]
        else: total[k] = new[k]
    return total

def stripZeros(histogram):
    for k in histogram.keys():
        if histogram[k] is 0:
            del histogram[k]
    return histogram

def normalizeAndLogHistogram(histogram, temp):
    total = 0
    for k in histogram.keys(): total += histogram[k]
    for k in histogram.keys(): histogram[k] /= float(total)
    for k in histogram.keys(): histogram[k] = -1 * math.log(histogram[k])
    return histogram

def raiseHistogram(height_offset,histogram):
    for k in histogram.keys():
        histogram[k] += height_offset
    return histogram


def stitchHistogram(TotalHistogram,Start,End,Interval):
    junctions = []
    offset = 0.0
    previous_value = 0.0
    switch = False
    # Bad code I use: junctions = [50, -50, 0]
    for i in range(0, UmbrellaWindows-2):
        junctions.append(Interval * i)
        junctions.append(-Interval * i )
    #print junctions
    # Go backwards...
```

```
    for m in range(Start,End+1)[::-1]:
        if m in TotalHistogram.keys():
            if switch:
                offset = (previous_value - TotalHistogram[m])
                switch = False
            TotalHistogram[m] = TotalHistogram[m] + offset
            previous_value = TotalHistogram[m]
            if m in junctions:
                switch = True
    return TotalHistogram


def PerformRun(n_steps,edge_length,ext_field,temp,start_flag=0,config_file=None):
    LatticeEdgeDimension = edge_length
    ExField = ext_field
    T = temp

    End = edge_length**2 #TODO: verify
    Start = -edge_length**2
    Interval = int(((End-Start)/UmbrellaWindows))
    TotalHistogram = {}
    tempAddHistogram = {}
    ProbabilityHistogram = {}
    for x in range(UmbrellaWindows):
        myStart = Start + (Interval * x)
        myEnd = min(Start + Interval * (x + 1), End)
        print myStart
        print myEnd
        for i in range(20):
            SpinLattice, energy, netmag =
initialize(start_flag,LatticeEdgeDimension,ExField,myStart,myEnd)
            SpinLattice, energy, netmag,histogram =
monte_carlo(n_steps,SpinLattice,T,energy,netmag,myStart,myEnd)
            ProbabilityHistogram = combineHistograms(histogram, ProbabilityHistogram)
            # Keeps track off all histograms samples from ONE umbrella region
            tempAddHistogram = combineHistograms(histogram, tempAddHistogram)

        # Strip zeroes to prevent math errors.  The rest is just calculating before
        tempAddHistogram = normalizeAndLogHistogram(stripZeros(tempAddHistogram),
temp)
        TotalHistogram = combineHistograms(tempAddHistogram, TotalHistogram)

        tempAddHistogram = {}

    before = TotalHistogram.copy()
    # Combine each new histogram
    TotalHistogram = stitchHistogram(TotalHistogram,Start,End,Interval)

    return SpinLattice, energy, netmag, ProbabilityHistogram, before, TotalHistogram

#Initialize, if given a start_flag of 0, initializes the lattice to be entirely
magnetized.
#If start_flag is 1, it reads from config_file for the lattice configuration

def initialize(start_flag,edge_length,field,myStart,myEnd):
    SpinLattice = [[0 for x in xrange(edge_length)] for y in xrange(edge_length)]
    netmag = 0
    energy = 0.
    ExField = field
    magTarget = 1
```

```
    #while magTarget % 2 != 0:
    magTarget = random.choice(range(myStart + EdgeBuffer,myEnd - EdgeBuffer))


    print "Mag target set..."
    print magTarget
    print "Generating magnetized configuration...\n"
    for x in xrange(edge_length):
        for y in xrange(edge_length):
            SpinLattice[x][y]=1
            netmag += 1

        energy = -2.0*(edge_length**2)

    flipsNeeded = (netmag - magTarget) / 2
    possibleSites = []
    for x in range(edge_length):
        for y in range(edge_length): possibleSites.append((x, y))
    random.shuffle(possibleSites)
    print len(possibleSites)
    print flipsNeeded
    for i in range(flipsNeeded):
        posTuple = possibleSites[i]
        SpinLattice[posTuple[0]][posTuple[1]] *= -1
        netmag += -2
        energy += 4.0

    print "Initial magnetization per spin = " + str(netmag/(edge_length**2)) + "\n"
    print "Initial energy per spin = " + str(energy/(edge_length**2)) + "\n"

    return SpinLattice,energy,netmag

#This method contains the bulk of the calculation.  Code that collects statistics,
however, is missing.
def monte_carlo(n_steps,spin,T,energy,netmag,start,end):

    histogram = {}
    avmag=0.0
    aven=0.0
    edge_length = len(spin[0])

    # with open("trajectory.dat",'w') as f:

    #Each 'step' of the algorithm attempts edge_length**2 trial moves.
    print "Generating trajectory...\n"
    for step in xrange(n_steps):
        for j in xrange(edge_length**2):
            x = int(edge_length*numpy.random.rand())
            y = int(edge_length*numpy.random.rand())

            spin,energy,netmag = trial_move(x,y,spin,energy,netmag,T,start,end)

            #if netmag == whatever m state you want:
            #    createSpinPlot(spin)

            if netmag not in histogram.keys():
                histogram[netmag] = 1
            else:
                histogram[netmag] += 1
            avmag = (avmag * (step) + netmag) / (step + 1) # step = (number of moves)
```

```
- 1
            aven  = (aven  * (step) + energy) / (step + 1)

        for val in range(start,end):
            if val not in histogram.keys():
                histogram[val] = 0 #This fills in missing values

        #/* Output averages */
        print "Average magnetization per spin = " + str(avmag/
(n_steps*edge_length**2)) + "\n"
        print "Average energy per spin = " + str(aven/(n_steps*edge_length**2)) + "\n"


    return spin, energy, netmag, histogram

#This method actually attempts the trial move, and decides whether to accept or reject
the trial.
def trial_move(x,y,spin,energy,netmag,T,start,end):
    neighbor_mag=0
    edge_length = len(spin[0])
    up,down,left,right = 0,0,0,0
    deltae = 0.0
    # if start == 200: end = 401
    magRange = range(start,end)

    if (x==0):
        left=spin[edge_length-1][y]
    else:
        left=spin[x-1][y]
    if (x==edge_length-1):
        right=spin[0][y]
    else:
        right=spin[x+1][y]
    if (y==edge_length-1):
        up=spin[x][0]
    else:
        up=spin[x][y+1]
    if (y==0):
        down=spin[x][edge_length-1]
    else:
        down=spin[x][y-1]

    #/* left, right, up, and down are the states */
    #/* of spins neighboring (x,y) */
    #/* compute change in energy if spin[x][y] were flipped, */
    my_old_spin = spin[x][y]
    neighbor_sum = up + down + left + right
    delta_E = 2 * ExField * my_old_spin + 2 * CC * my_old_spin * neighbor_sum
    #/*************************************************************/

    #/* accept according to Metropolis Monte Carlo rules */
    #/* update magnetization and energy if necessary */
    if ((netmag - 2 * my_old_spin) in magRange and min(1, math.exp(-1 * delta_E / T))
> random.random() ):
        # Flip the spin if min(1, e^(-beta E)) > random float (from [0, 1))
        energy += delta_E
        spin[x][y] = -1 * my_old_spin
        netmag -= (2 * my_old_spin)
    #/*************************************************************/
    return spin, energy, netmag
```

```
#Writes a configuration to file.  Spins are written as integers, tab delimited.
def write_config(filnam,spin):
    with open(filnam,'w') as f:
        for x in xrange(len(spin[0])):
            f.write(str("\t".join([str(s) for s in spin[x]])) + "\n")

#Reads from a configuration file.  Reads from the same format as the write_config
method above--i.e. tab delimited integers, with an equal
#number of rows as columns.
def read_config(filnam):
    spin = []
    with open(filnam,'r') as f:
        for line in f:
            ThisRow = []
            for s in line.split("\t"):
                ThisRow.append(int(s))
            spin.append(ThisRow)

    return spin

def createSpinPlot(SpinLattice):
    edge_length = len(SpinLattice)
    SpinColor = [[0 for x in xrange(edge_length)] for y in xrange(edge_length)]
    for y in range(edge_length):
        for x in range(edge_length):
            if SpinLattice[x][y] == 1:
                SpinColor[x][y] = [1.0, 1.0, 1.0]
            else:
                SpinColor[x][y] = [0.0, 0.0, 1.0]
    SpinArray = numpy.array(SpinColor)
    plt.imshow(SpinArray, interpolation='nearest')
    plt.show()
    plt.clf()

#This is how you might call the program to perform 10000 steps in a 20x20 lattice at 0
applied field and at 1.0 K, where the lattice is initially fully magnetized

tempPoints = []
amagPoints = []
temp = 40
step = 1

SpinLattice, energy, netmag, ProbabilityHistogram, histogram, histogramAfter =
PerformRun(10,20,0,temp)
HistogramPoints = []
MagPoints = []

ProbabilityHistogramPoints = []
PMagPoints = []

ProbabilityHistogram = stripZeros(ProbabilityHistogram)

for x in range(-400,401):
    if x in ProbabilityHistogram.keys():
        ProbabilityHistogramPoints.append(ProbabilityHistogram[x])
        PMagPoints.append(x)
        print "At %i, %f" % (x, ProbabilityHistogram[x])
```

```
plt.plot(PMagPoints,ProbabilityHistogramPoints,'ro')
plt.show()
plt.clf()

for x in range(-400,401):
    if x in histogram.keys():
        HistogramPoints.append(histogram[x])
        print "At %i, %f" % (x, histogram[x])
        MagPoints.append(x)

plt.plot(MagPoints,HistogramPoints,'ro')
plt.show()
plt.clf()

OtherMagPoints = []
OtherHistogramPoints = []

current_min = 90
current_min_m = 1000

for x in range(-400,401):
    if x in histogramAfter.keys():
        #current_min = min(current_min, histogramAfter[x])
        #current_min_m = x
        #Use this code to get the Beta * A(M) for the 2nd derivative.
        OtherHistogramPoints.append(histogramAfter[x])
        OtherMagPoints.append(x)
        print "At %i, %f" % (x, histogramAfter[x])
plt.plot(OtherMagPoints,OtherHistogramPoints,'ro')
plt.show()

#print current_min
#print current_min_m

#This is how you might call a program to perform the same, except on a user-specified
input file:
#Note that when using a user-specified input file, the edge dimension argument is
ignored (in this case, 20 is ignored, and the
#edge dimension is set to the dimensions of the input file lattice
#PerformRun(10000,20,0.0,1.0,1,"startconfig.in")
```