Jacob Dennis

Physics 411

18 April 2016

Prof. Gull

Midterm IV

**The Ising Model in Two Dimensions**

*Code*

import numpy as np

import numpy.random as rand

import matplotlib.pyplot as plt

def MakeLattice(L):

A = np.ones([L, L])

return A

def Hamiltonian(A):

L = A.shape[0]

E = 0.

for i in range(L):

for j in range(L):

E += -A[i][j] \* A[(i + 1) % L][j]

E += - A[i][j] \* A[i][(j + 1) % L]

return E

def step(A, inverse\_T):

L = A.shape[0]

x = rand.random\_integers(0, int(L) - 1)

y = rand.random\_integers(0, int(L) - 1)

dE = EnergyDifference(A, x, y)

if (min(1, np.exp(-dE \* inverse\_T)) > rand.random()):

A[x][y] \*= -1

return None

def EnergyDifference(A, x, y):

E\_old = Hamiltonian(A)

A[x][y] \*= -1.

E\_new = Hamiltonian(A)

A[x][y] \*= -1.

return E\_new - E\_old

def Magnetization(A):

L = A.shape[0]

return np.sum(A) / (L\*\*2)

def AverageMagnetizations(A, T, nsteps):

abs\_m\_average = 0.0

m2\_average = 0.0

for i in range(nsteps):

step(A, 1. / T)

abs\_m\_average += np.abs(Magnetization(A))

m2\_average += (Magnetization(A)\*\*2)

print '< |m| >:', abs\_m\_average / nsteps, '\n< m^2 >:', m2\_average / nsteps

return None

def PlotALattice(A, filename, plotttitle):

L = A.shape[0]

x, y = np.mgrid[slice(0, L + 1, 1), slice(0, L + 1, 1)]

plt.figure(0)

plt.clf()

plt.pcolor(x, y, A, cmap = 'RdBu', vmin = -1., vmax= 1.)

plt.axis([x.min(), x.max(), y.min(), y.max()])

plt.xlabel('x')

plt.ylabel('y')

plt.title(plotttitle)

plt.colorbar()

plt.savefig(filename)

return None

def main():

A = MakeLattice(10)

print 'Ground State Energy:', Hamiltonian(A), '\n'

#Part 1 - T >> Tc

print 'T = 10000'

AverageMagnetizations(A, 10000., 1000)

print 'Energy:', Hamiltonian(A), '\n'

PlotALattice(A, 'Midterm 4 - High Temperature Lattice.png', 'High Temperature Spin States')

#Part 2 - T << Tc

A = MakeLattice(10)

print 'T = 1'

AverageMagnetizations(A, 1., 1000)

print 'New Energy:', Hamiltonian(A), '\n'

PlotALattice(A, 'Midterm 4 - Low Temperature Lattice.png', 'Low Temperature Lattice')

#Part 4 - m(T) and E(T)

#6x6

dimensionsRange = [6, 8, 10]

plt.figure(1)

plt.clf()

plt.figure(2)

plt.clf()

for d in dimensionsRange:

TRange = np.arange(.1, 100.)

mRange = []

ERange = []

URange = []

for T in TRange:

A = MakeLattice(d)

for i in range(1000):

step(A, 1. / T)

ERange.append(Hamiltonian(A) / (d\*\*2))

m = Magnetization(A)

mRange.append( m / (d\*\*2))

plt.figure(1)

plt.plot(TRange, ERange, label = str(d) + 'x' + str(d))

plt.figure(2)

plt.plot(TRange, mRange, label = str(d) + 'x' + str(d))

plt.figure(1)

plt.xlabel('T')

plt.ylabel('Energy')

plt.legend(loc = 0)

plt.title('Energy per site')

plt.savefig('Midterm 4 - Energies')

plt.figure(2)

plt.xlabel('T')

plt.ylabel('Magnetization')

plt.title('Magnetization per site')

plt.legend(loc = 0)

plt.savefig('Midterm 4 - Magnetizations')

if \_\_name\_\_ == '\_\_main\_\_':

main()

*Results*

Ground State Energy: -200.0

T = 10000

< |m| >: 0.13392

< m^2 >: 0.0425792

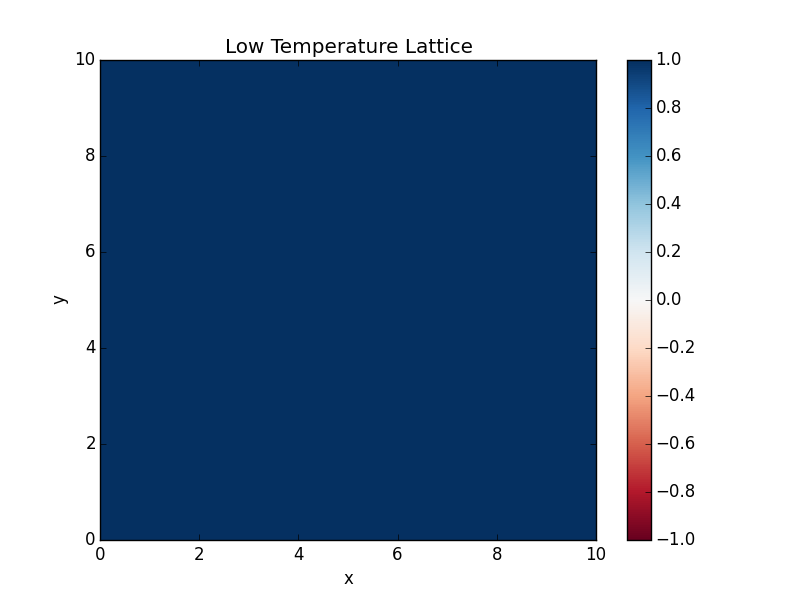
Energy: 12.0

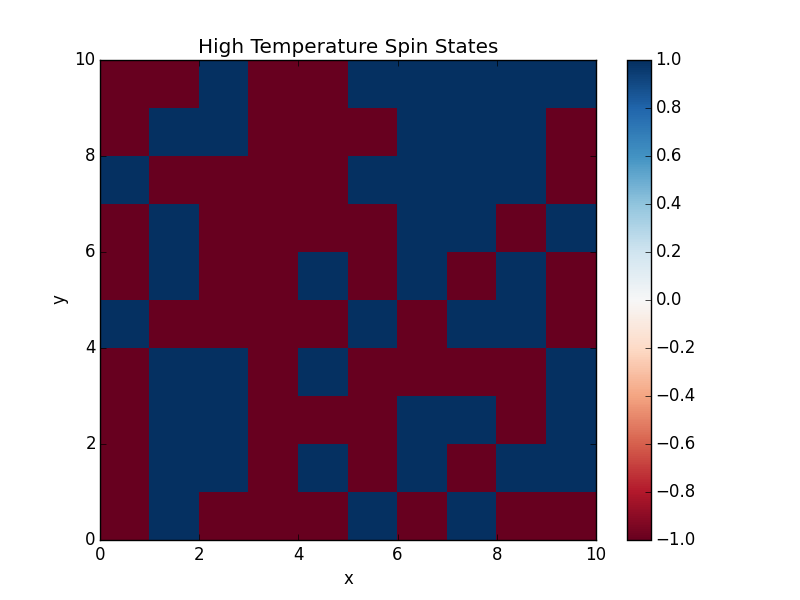
T = 1

< |m| >: 0.99808

< m^2 >: 0.9961984

New Energy: -200.0

****



**Cluster Labeling Algorithm**

*Code*

import numpy as np

import numpy.random as rand

import matplotlib.pyplot as plt

def MakeLattice(L, typeOfLattice):

A = np.zeros((L, L), dtype = typeOfLattice)

return A

def FirstRow(A, A\_Labeled, labelsIndex, LabelsList):

L = A.shape[0]

if A[0][0] == 0:

A\_Labeled[0][0] = 0.

else:

A\_Labeled[0][0] = labelsIndex

labelsIndex += 1

LabelsList.append(labelsIndex)

for j in range(1, L):

if A[0][j] == 0:

A\_Labeled[0][j] = 0.

else:

if A[0][j - 1] == 0.:

A\_Labeled[0][j] = labelsIndex

labelsIndex += 1

LabelsList.append(labelsIndex)

else:

A\_Labeled[0][j] = LabelsList[int(A\_Labeled[0][j - 1])]

return labelsIndex

def FirstColumn(A, A\_Labeled, labelsIndex, LabelsList):

L = A.shape[1]

for i in range(1, L):

if A[i][0] == 0:

A\_Labeled[i][0] = 0.

else:

if A[i - 1][0] == 0.:

A\_Labeled[i][0] = labelsIndex

labelsIndex += 1

LabelsList.append(labelsIndex)

else:

A\_Labeled[i][0] = LabelsList[int(A\_Labeled[i - 1][0])]

return labelsIndex

def OccupiedNeighbors(A, i, j, A\_Labeled, LabelsList):

if (A[i - 1][j] != 0) & (A[i][j - 1] == 0):

A\_Labeled[i][j] = LabelsList[int(A\_Labeled[i - 1][j])]

elif (A[i - 1][j] == 0) & (A[i][j - 1] != 0):

A\_Labeled[i][j] = LabelsList[int(A\_Labeled[i][j - 1])]

else:

if LabelsList[int(A\_Labeled[i - 1][j])] == LabelsList[int(A\_Labeled[i][j - 1])]:

A\_Labeled[i][j] = A\_Labeled[i - 1][j]

else:

ConflictCase(A, i, j, A\_Labeled, LabelsList)

return A\_Labeled

def ConflictCase(A, i, j, A\_Labeled, LabelsList):

bigger = int(np.amax([A\_Labeled[i - 1][j], A\_Labeled[i][j - 1]]))

smaller = int(np.amin([A\_Labeled[i - 1][j], A\_Labeled[i][j - 1]]))

LabelsList[bigger] = LabelsList[smaller]

A\_Labeled[i][j] = smaller

return A\_Labeled

def RelabelMatrix(A, LabelsList):

L = np.shape(A)[0]

for i in range(L):

for j in range(L):

A[i][j] = LabelsList[int(A[i][j])]

return np.float32(A)

def RelabelList(myList):

for i in range(1, len(myList)):

if (myList[i] - myList[i - 1]) <= 0.:

for j in range(i + 1, len(myList)):

if myList.index(myList[j]) == j:

myList[j] -= 1.

return myList

def HoshenKopelman(A):

L = A.shape[0]

A\_Labeled = np.zeros((L, L), dtype = 'float')

LabelsList = [0, 1]

labelsIndex = 1

labelsIndex = FirstRow(A, A\_Labeled, labelsIndex, LabelsList)

labelsIndex = FirstColumn(A, A\_Labeled, labelsIndex, LabelsList)

for i in range(1, L):

for j in range(1, L):

if A[i][j] == 0.:

A\_Labeled[i][j] = 0

elif (A[i - 1][j] == 0) & (A[i][j - 1] == 0):

A\_Labeled[i][j] = labelsIndex

labelsIndex += 1

LabelsList.append(labelsIndex)

else:

OccupiedNeighbors(A, i, j, A\_Labeled, LabelsList)

RelabelList(LabelsList)

RelabelMatrix(A\_Labeled, LabelsList)

return A\_Labeled

def FillPointsInLattice(A, points):

for point in points:

A[point] = 1.

return A

def PlotLabeledLattice(A\_Labeled, filename, figuretitle):

L = A\_Labeled.shape[0]

x, y = np.mgrid[slice(0, L + 1, 1), slice(0, L + 1, 1)]

plt.figure(0)

plt.clf()

plt.pcolor(x, y, A\_Labeled, cmap = 'Paired', vmin = 0., vmax = np.amax(A\_Labeled))

plt.axis([x.min(), x.max(), y.min(), y.max()])

plt.colorbar()

plt.xlabel('x')

plt.ylabel('y')

plt.title(figuretitle)

plt.savefig(filename)

return None

def OccupyALattice(L, p, typeOfLattice):

A = MakeLattice(L, typeOfLattice)

for i in range(L):

for j in range(L):

if rand.random() > p:

A[i][j] = 0

else:

A[i][j] = 1

return A

def main():

setOfPoints1 = [(0, 6), (1, 0), (1, 1), (1, 2), (1, 3), (1, 5), (1, 6), (2, 5), (3, 1), (3, 2), (3, 3),

(4, 3), (5, 0), (5, 1), (5, 5), (5, 6), (6, 0), (6, 1), (6, 4), (6, 5)]

setOfPoints2 = [(1, 5), (2, 1), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 3), (4, 6), (5, 0), (5, 6),

(5, 3), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6)]

Config1 = FillPointsInLattice(MakeLattice(7, 'int'), setOfPoints1)

Config2 = FillPointsInLattice(MakeLattice(7, 'int'), setOfPoints2)

LabeledConfig1 = HoshenKopelman(Config1)

LabeledConfig2 = HoshenKopelman(Config2)

print 'Test Configuration 1:\n', LabeledConfig1

print 'Test Configuration 2:\n', LabeledConfig2

A = OccupyALattice(100, .58, 'int')

B = HoshenKopelman(A)

PlotLabeledLattice(B, 'Midterm 4 - 100x100 Hoshen-Kopelman.png', '100 x 100 Lattice, Grouped by Hoshen-Kopelman')

if \_\_name\_\_ == '\_\_main\_\_':

main()

*Results*

Test Configuration 1:

[[ 0. 0. 0. 0. 0. 0. 1.]

[ 2. 2. 2. 2. 0. 1. 1.]

[ 0. 0. 0. 0. 0. 1. 0.]

[ 0. 4. 4. 4. 0. 0. 0.]

[ 0. 0. 0. 4. 0. 0. 0.]

[ 3. 3. 0. 0. 0. 5. 5.]

[ 3. 3. 0. 0. 5. 5. 0.]]

Test Configuration 2:

[[ 0. 0. 0. 0. 0. 0. 0.]

[ 0. 0. 0. 0. 0. 1. 0.]

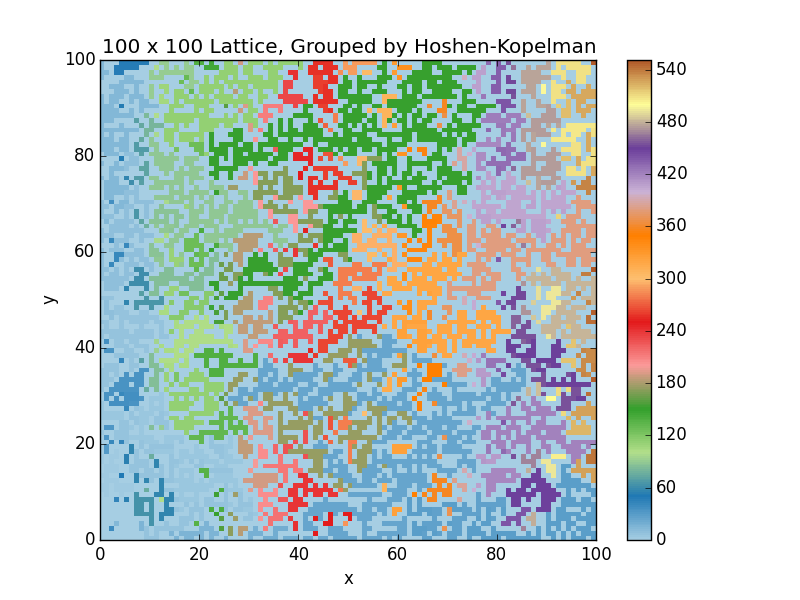
[ 0. 1. 0. 0. 0. 1. 0.]

[ 0. 1. 1. 1. 1. 1. 0.]

[ 0. 0. 0. 1. 0. 0. 1.]

[ 1. 0. 0. 1. 0. 0. 1.]

[ 1. 1. 1. 1. 1. 1. 1.]]



**The Percolation Problem**

*Code*

import numpy as np

import numpy.random as rand

from HoshenKopelman import HoshenKopelman #.py file containing all the programs of Problem 2

from HoshenKopelman import OccupyALattice

import matplotlib.pyplot as plt

def CheckIfArrayContains(array, n):

for i in array:

if i == n:

return True

return False

def CheckPercolation(A):

L = A.shape[0]

clustersInTop = []

clustersInBottom = []

for i in range(L):

if A[0][i] != 0:

clustersInTop.append(A[0][i])

if A[L - 1][i] != 0:

clustersInBottom.append(A[L - 1][i])

for cluster in clustersInTop:

if CheckIfArrayContains(clustersInBottom, cluster):

return True

return False

def P(p, d):

numPercolating = 0.

numTries = 5000

for n in range(numTries):

A = OccupyALattice(d, p)

LabeledA = HoshenKopelman(A)

if CheckPercolation(LabeledA):

numPercolating += 1

return numPercolating / numTries

def main():

pRange = np.linspace(0.0, 1.0, 50)

PRange\_2 = []

PRange\_3 = []

PRange\_4 = []

for p in pRange:

PRange\_2.append(P(p, 2))

PRange\_3.append(P(p, 3))

PRange\_4.append(P(p, 4))

counter += 1

plt.figure(0)

plt.clf()

plt.plot(pRange, PRange\_2, label = '2x2')

plt.plot(pRange, PRange\_3, label = '3x3')

plt.plot(pRange, PRange\_4, label = '4x4')

plt.legend(loc = 0)

plt.title('P(p)')

plt.xlabel('p')

plt.ylabel('P')

plt.savefig('Midterm 4 - Percolation.png')

plt.figure(1)

plt.clf()

plt.plot(pRange, np.array(PRange\_2) - pRange, label = '2x2')

plt.plot(pRange, np.array(PRange\_3) - pRange, label = '3x3')

plt.plot(pRange, np.array(PRange\_4) - pRange, label = '4x4')

plt.legend(loc = 0)

plt.title('P(p) - p')

plt.xlabel('p')

plt.ylabel('P')

plt.savefig('Midterm 4 - Percolation 2.png')

if \_\_name\_\_ == '\_\_main\_\_':

main()

*Results*

