2D Ising Model Simulation

Jim Ma

Department of Physics

jma@physics.ucdavis.edu

Abstract: In order to simulate the behavior of a ferromagnet, I used a simplified 2D Ising model. This model is based on the key features of a ferromagnet and the Metropolis algorithm. The whole model is implemented in Python. We can examine how the temperature affects the phase transition of ferromagnet generated by executing this simulation.

Introduction

In order to simulate the behavior of a ferromagnet, we examine the features of a ferromagnet first. Since there are billions of atomic dipoles in an ordinary size of a ferromagnet, even the best computer is impossible to take it. We use Monte Carlo summation, which generates a random sampling, and Metropolis algorithm, which low-energy states occur more often than high-energy state, to build the Ising model and

implement this model in Python.

Features of a Ferromagnet

In a ferromagnet, there are an enormous number of small patches called domains in which the atomic dipoles are aligned. Since the domains are randomly oriented, there is no large scale magnetization. Accordingly, raising the temperature will increase the random fluctuations and destroy the alignment of the atomic dipoles. To every ferromagnet, at certain temperature, called Curie temperature, the magnetization becomes zero. A ferromagnet will become a paramagnet as the temperature is above the Curie temperature.

Ising Model

To simply our model, we assume:

1. There are N atomic dipoles located on the N sites of a ferromagnet.
2. Each atomic dipole can be in one of the two possible states, called *spin (S)*, $S = \pm 1$ (spin up: 1, spin down: -1).
3. The total energy of a ferromagnet is $E = -J\sum S_i S_j$, J is a constant and the sum is over all pairs of adjacent spins

From assumption 3, the energy of two neighboring pairs is $-E$ if they are parallel and $+E$ if they are antiparallel.

The key point of the Metropolis algorithm is to use the Boltzmann factors as a guide to generate the random sampling of states. It starts with any state randomly. Calculate the energy of the state, ΔU. If $\Delta U \leq 0$, the system's energy will decrease or remain unchanged after the state flipped, then go and change the state. If $\Delta U > 0$, compare the probability, $\exp(-\Delta U/kT)$, of the flip to the probability generated at random. If there is no flip of the state, then there is no change of the system. This process is repeated over and over again until every state has chances to flipped.

Implementation

The software code is implemented in Python and listed at the end of this report. The program defines a size of n×n domain and a two-dimensional array state(i,j) for the spin orientations. The size of the domain, temperature, size of the cell for dipole, and the number of

time steps can be changed for different run. The program uses subroutine, Initialize, to define the state of the system randomly.

The heart of the program is the "Main loop," which executes the Metropolis algorithm. Within the loop, we first choose a state randomly by using the *random*(1), which returns a value between 0 and 1. Then, use the subroutine, dU, to calculate the ΔU. In the subroutine, dU, we implement the Mean Field Approximation method:

$E = -JnŜ$, n : number of nearest neighbors (n=4 in 2_D)

Ŝ : average alignment of neighbors

and periodic boundary conditions, which we wrap around the states at the boundary (edge). So that the right edge is immediately left of the left edge and the bottom edge is immediately above the top edge.

At the end of the loop, we color the cell whose spin has flipped by using the subroutine, ColorCell.

Conclusion

If you like to find the Curie temperature of a ferromagnet, you have to set different T (temperature) values for different runs and compare their results. If the T we set is below the actual Curie temperature, you'll get the final picture is either totally red or totally blue. Then, you can try a higher T for the next run. Repeat this process until you get the best Curie temperature.

How many iterations are enough to get the right result? This depends on the domain size you define at the beginning of the run. Since the Ising model is based on random sampling, I set a infinite loop for iteration. Leave the user to decide when to stop execution!

Bibliography

Daniel V. Schroeder, "An Introduction to Thermal Physics," by Addison Wesley Longman.

Program: 2D Ising Model Simulation

```
#Phy250 Project: Ising Model Simulator  by Jim Ma
#
# TwoDIsing.py: Two-dimensional Ising Model simulator
#       Using PyGame/SDL graphics
#
"""
 Options:
 -t #  Temperature
 -n #  Size of lattice
 -c #  Size of cell
 -s #  Number of time steps
"""

from numpy import *
from numpy.random import *
import getopt,sys
import time
from random import randint
try:
        import Numeric as N
        import pygame
        import pygame.surfarray as surfarray
except ImportError:
        raise ImportError, "Numeric and PyGame required."

#
#               Ising Model routines                #
#
# Initialize crystal lattice, load in initial configuration
def Initialize(nSites):
```

```
        state = N.zeros((nSites,nSites))
        for i in range(nSites):
          for j in range(nSites):
            if randint(0,1) > 0.5:     # dipole spin up
               state[i][j] = 1
            else :  state[i][j] = -1         # dipole spin down

        return state

# Use Mean Field approximation and periodic BC to compute
# dU for decision to flip a dipole
#
def dU(i, j, nSites, state):
  m = nSites - 1
  if i == 0 :               # state[0,j]
    top = state[m,j]
  else :
    top = state[i-1,j]

  if i == m :                # state[m,j]
    bottom = state[0,j]
  else :
    bottom = state[i+1,j]

  if j == 0 :               # state[i,0]
    left = state[i,m]
  else :
    left = state[i,j-1]

  if j == m :                # state[i,m]
    right = state[i,0]
  else :
    right = state[i,j+1]

  return 2.*state[i,j]*(top+bottom+left+right)

# Color the cell based on dipole direction:
#     Dipole spin is up : color cell red
```

```
#     Dipole spin is down : color cell blue
#
def ColorCell(state,i,j):
   if state[i][j] > 0:    # dipole spin is up
          screen.fill(UpColor,[i*CellSize,j*CellSize,CellSize,CellSize])
   else :
screen.fill(DownColor,[i*CellSize,j*CellSize,CellSize,CellSize])

# Set defaults
T = 2  # Temperature
nSites   = 30 #30
CellSize = 24  #16
nSteps = 1000


# Get command line arguments, if any
opts,args = getopt.getopt(sys.argv[1:],'t:n:c:s:')
for key,val in opts:
        if key == '-t': T        = int(val)
        if key == '-n': nSites   = int(val)
        if key == '-c': CellSize = int(val)
        if key == '-s': nSteps   = int(val)

print 'T = ', T
print 'nSites = ', nSites
print 'CellSize = ', CellSize
print 'nSteps = ',nSteps

size = (CellSize*nSites,CellSize*nSites)
        # Set initial configuration
state = Initialize(nSites)

pygame.init()
UpColor = 255, 0, 0       # red
DownColor = 0, 0, 255     # blue
        # Get display surface
screen = pygame.display.set_mode(size)
pygame.display.set_caption('2D Ising Model Simulator')
```

```
        # Clear display
screen.fill(UpColor)
pygame.display.flip()
        # Create RGB array whose elements refer to screen pixels
sptmdiag = surfarray.pixels3d(screen)

   # display initial dipole configuration
for i in range(nSites):
  for j in range(nSites):
    if state[i][j] < 0:

screen.fill(DownColor,[i*CellSize,j*CellSize,CellSize,CellSize])

t = 0
t0 = time.clock()
   # total execution time
t_total = time.clock()

# Main loop
flag = 1
while flag > 0:
        for event in pygame.event.get():
           # Quit running simulation
                if event.type == pygame.QUIT: sys.exit()
                # randomly select cell
       i = int(random(1)*nSites)
        j = int(random(1)*nSites)
        # Any system energy change if flip dipol
        dE = dU(i,j,nSites,state)
        # flip if system will have lower energy
        if dE <= 0. :
         state[i][j] = -state[i][j]
         ColorCell(state, i, j)
       # otherwise do random decision
       elif random(1) < exp(-dE/T) :
           state[i][j] = -state[i][j]
           ColorCell(state, i, j)
```

```
        pygame.display.flip()
        t += 1
        if (t % nSteps) == 0:
                t1 = time.clock()
                if (t1-t0) > 0.001 :
                   print 't1 = ', t1
                   print "Iterations per second: ", float(nSteps) / (t1 - t0)
                t0 = t1
        # calculate execution time
        dt = time.clock()-t_total
        if dt > 200.0 : flag = -1
#matshow(state)
print "Total simulation time is %g seconds of temperature %g K" %
(dt,T)
```