```python
# -*- coding: utf-8 -*-
"""
Created on Sun Jan  5 11:47:14 2020

@author: Alexm
"""

import matplotlib
#matplotlib.use("Qt4Agg") #take this out on jupyter
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from numpy.random import rand
%matplotlib notebook
print('Alexandra Mulholland 17336557')
print('Part 4: Animation of the configurations')

#Part 4- Simulation of the configuration of the matrix
#We will be using a slightly different Metropolis algorithm for the sake of simplicity
#For a ferromagnetic material at constant temperature and no external magentic field influence


#Here, as before, we will randomly select a point, s, in the lattice and identify the 4 neighbours to said point.

#The spin product is again calculated- although this time outside of the definition for the energy change (EC), unlike in the
#codes for parts 1,2 and 3.

#ec is the spin product (spinproduct) and exchange energy (J) product, which is later used to find the energy change EC upon a lattice point's spin
#being flipped.

#ec is used in the Metropolis algorithm, which has been altered from parts 1,2 and 3.

#Initial configuration is produced: an N x N lattice of spins being either 1 or -1

def randomstate(N):
    state = 2*np.random.randint(2, size=(N,N))-1
    return state

def Ising(a,b):
    s = config[a,b]
    naybor = config[(a+1)%N,b] + config[a,(b+1)%N] + config[(a-1)%N,b] + config[a,(b-1)%N]
    spinproduct = s * naybor
    ec = -J*spinproduct
    return ec
#The Metropolis algorithm (altered). Rather than elaborating on this within the report, I will simply
#explain the differences in said algorithm here.
    #A random point within the lattice is selected
    #The spin of said point is determined and flipped regardless of energy change magnitude
    #The energy change before and after the flip is calculated (finish-start)
    #As with the previous codes, a random number between 0 and 1 is generated
    #If this number is greater than the Boltzmann factor, the spin is then flipped again
    #This contrasts again to previous codes where if this number was LESS than the Boltzmann factor, the spin was flipped
    #This change still makes logical sense: This is due to the fact that the spin had been
    #flipped regardless of what was energetically favourable

#"The single asterisk form (*args) is used to pass a non-keyworded, variable-length argument list"
#Without this, the animation would not change. It calls the parameters listed underneath
def moncar(*args):
    a = np.random.randint(0,N) #random lattice point is chosen, coordinate (a,b)
    b = np.random.randint(0,N)
    start = Ising(a,b) #its spin product with its neighbours is found and multiplied by the exchange energy
    config[a,b] *= -1 #flipping the spin of chosen point
    finish = Ising(a,b) #Final spin of chosen point x -J x spin of neighbours
    EC = finish-start #energy change as a result of the flipped spin
    if(rand() > np.exp(-EC/T)): #If this energy change results in the random number generated being greater than the boltzmann constant,
        config[a,b] *= -1 #the spin is flipped again (back to its original spin). Otherwise, it remains in its flipped spin state
    mesh.set_array(config.ravel()) #returns the array, flattened. "1D array with all the input-array elements and
```

```
with the same type as it"
    return config



T = 1.0 #Temperature-- again, 1/T=beta
N = 20 #NxN matrix
J = 1 #Exchange energy
stepsmoncar = 2000 #Number of times the Metropolis algorithm will be executed
config = randomstate(N) #Initial, random matrix of spins

#Performing the simulation
fig = plt.figure(figsize=(10, 10), dpi=70) #figure size and dpi is how many pixels the figure comprises
fig.suptitle("Animation of the execution of the Metropolis algorithm for a 20x20 lattice", fontsize=15)
plt.title("For T = %0.1fK" %T, fontsize=10)
plt.gca().axes.get_yaxis().set_visible(False)
plt.gca().axes.get_xaxis().set_visible(False)
X, Y = np.meshgrid(range(N), range(N)) #grid size
mesh = plt.pcolormesh(X, Y, config, cmap = plt.cm.winter, vmin=-1, vmax=1) #vmin and vmax define the data range
k = animation.FuncAnimation(fig, moncar, frames = stepsmoncar, interval = 5, blit = True)
plt.show()
```