```
'''
Spin glass optimization problem
Assume neighbor to neighbor coupling and find minimum energy of the spin glass
system
Simulated annealing Monte Carlo scheme
'''
from random import random,uniform,choice,randrange
import numpy as np
from sympy import *
from scipy.optimize import *
import time
import matplotlib.pyplot as plt
from math import exp

#create a dictionary to store each minimum value iterations
#initially set all minima values to zero
minima={}
for m in np.arange(0,-100,-.1):
    minima[round(m,1)]=0

#run program at least 30 times
#define size of grid; only square grids allowed
s=4
energy=[]
avg_times=[]

#return bonds between spins J matrix; neighbor-to-neighbor coupling
#Neighbor-to-neighbor coupling bonds
def create_bonds(s):
    J={}
    #horizontal bonds
    c=0
    for j in range(0,s):
        for i in range(0,s-1):
            J[i+c,i+c+1]=uniform(-1,1)
        c+=s
    #vertical bonds
    for i in range(0,s):
        for j in range(0,s**2-s,s):
            J[j+i,j+i+s]=uniform(-1,1)
    return J
#All-neighbor coupling bonds
def create_bonds_all(s):
    J={}
    for i in range(1,s**2):
        for j in range(2,s**2+1):
            if i!=j and (j,i) not in J:
                J[i,j]=uniform(-1,1)
    return J
J=create_bonds_all(s)

#extract the keys of the dictionary J as type integer
spins=list(J.keys())


#Find H (energy) by adding each spin combination with its J value
def find_H(phi):
    H=0
    for sp in spins:
```

```
            H+=-J[sp]*cos(phi[sp[0]]-phi[sp[1]])
        #H+=-J[sp]*cos(phi[:sp[0]]-phi[:sp[1]])
    return H

'''*************************************************************************'''
#monte-carlo step
def monte_carlo(find_H,t,emin,phi0):
    for n in range(0,s**2):
        e0=find_H(phi0)
        if e0<emin:
            emin=e0
    #alter a spin at random by a random amount between -1 and 1
        k=randrange(s**2)
        phi=phi0[k]
        r=uniform(-1,1)
        phi0[k]+=.025*(-1+2*r)
        #accept or reject the move
        e1=find_H(phi0)
        en=round((e1-e0)/t,10)
        if 1e-150<=en<=1e150:
            if random()>exp(-en):
                phi0[k]=phi
    return emin

'''*************************************************************************'''
phi_start=[uniform(0,2*np.pi) for i in range(0,s**2+1)] #remember to delete +1 for
neighbor-to-neighbor coupling
print(phi_start)
#main program
for n in range(0,50):
    times=[]
    energies=[]
    emin=10000
    t=10
    #phi0=phi_start
    phi0=[1.5118887800682619, 0.16487414752053156, 4.774015187098554,
2.9742638116465896, 0.7994055447646006, 2.018131004605727, 4.815092693443751,
5.965179214314855, 2.7458258407519875, 5.168806824227143, 2.9817126098669684,
1.23293523529134, 5.940312342221067, 4.130048942375131, 5.789019840334188,
3.397901359344739, 5.739689875028841]
    start=time.time()
    while .0001<=t<=10:
        m=monte_carlo(find_H,t,emin,phi0)
        t=t*.975
        times.append(1/t)
        energies.append(m)
    end=time.time()
    num=round(m,1)
    minima[num]+=1
    avg_times.append(end-start)
    energy.append(m)
    print('Iteration: {0}  Min energy: {1}  time taken: {2}'.format(n+1,m,end-
start))
    #plot energy vs. time for debugging
    #plt.plot(times,energies)
    #plt.title('Time vs. Energy, Annealing 3x3 grid')
    #plt.show()
average=sum(avg_times)/len(avg_times)
print('Average time per iteration: {}'.format(average))
```

```
numbers=list(minima.keys())
mins=list(minima.values())
plt.plot(numbers,mins,color='red')
plt.suptitle('Monte-Carlo Annealing Distribution of Min Energy All-Neighbor')
plt.title('4x4 grid')
plt.xlabel('Minimum energy values')
plt.ylabel('Frequency of occurance')
plt.xlim([-10,-25])
plt.show()


#calling function for algorithm that checks after each iteration if maximum time
#has been reached. If it has, the algorithm stops and returns its current minimum.
#The callback function for differential evolution method

def calling1(H,convergence):
    max_sec=7
    elapsed=time.time()-start
    if elapsed>max_sec:
        convergence=1.1
        #print(elapsed)
        return True
    else:
        print('the value is {0}'.format(H))


#callback function for basin hopping method
def calling2(phi,H,accepted):
    max_sec=7
    elapsed=time.time()-start
    if elapsed>max_sec:
        #print(elapsed)
        return True
    else:
        print('the value is {0} accepted {1}'.format(H,accepted))


#other two methods for comparison
bounds=[(0,2*np.pi) for i in range(0,s**2+1)]
start=time.time()
res=differential_evolution(find_H,bounds,maxiter=10,callback=calling1)
print('differential evolution: {0} time taken: {1}'.format(res.fun,time.time()-
start))
start=time.time()
res=basinhopping(find_H,phi0,niter=1,callback=calling2)
print('basinhopping: {0} time taken: {1}'.format(res.fun,time.time()-start))

'''-------------------------------------------------------------------------'
''
'''
CFD solution for incompressible one-dimensional Coutte flow between two plates
using implicit Crank-Nicolson technique and Thomas' algorithm to solve the tri-
diagonal
matrix.
'''
import matplotlib.pyplot as plt
import numpy as np
```

```
#Thomas'
algorithm----------------------------------------------------------------------
------------------------------------------------------
'''Uses Thomas algorithm for solving a tridiagonal matrix for n unknowns.
a, b, and are a list of the matrix entries
Matrix form of:
[b1 c1 ] [x1] [d1]
|a2 b2 c2 | |x2| |d2|
| a3 b3 c3 | |x3| |d3|
| | |' |= | '|
| | |' | | '|
[ an bn cn] |xn] [dn]
'''
def thomas(a,b,c,d):
    #find size of matrix and determine n
    n=len(b)
    #convert to float
    for i in range(0,len(a)):
        a[i]=float(a[i])
    for i in range(0,len(b)):
        b[i]=float(b[i])
    for i in range(0,len(c)):
        c[i]=float(c[i])
    for i in range(0,len(d)):
        d[i]=float(d[i])

    p=[]
    q=[]
    p.append(c[0]/b[0])
    q.append(d[0]/b[0])
    for j in range(1,n):
        pj=c[j]/(b[j]-a[j-1]*p[j-1]+.000001)
        qj=(d[j]-a[j-1]*q[j-1])/(b[j]-a[j-1]*p[j-1]+.000001)
        p.append(pj)
        q.append(qj)
    x=[]
    x.append(q[n-1])
    for j in range(n-2,-1,-1):
        xj=q[j]-p[j]*x[0]
        x.insert(0,xj)
    return x
#get new K values
def K_values(a,b,c,d):
    n=len(b)
    #convert to float
    for i in range(0,len(a)):
        a[i]=float(a[i])
    for i in range(0,len(b)):
        b[i]=float(b[i])
    for i in range(0,len(c)):
        c[i]=float(c[i])
    for i in range(0,len(d)):
        d[i]=float(d[i])

    p=[]
    q=[]
    p.append(c[0]/b[0])
    q.append(d[0]/b[0])
    for j in range(1,n):
```

```
        pj=b[j]-(a[j]*c[j-1])/p[j-1]
        qj=d[j]-(q[j-1]*a[j])/p[j-1]
        p.append(pj)
        q.append(qj)
    return q
#get main diagonal coefficients for next time step
def diagonals(a,b,c,d):
    n=len(b)
    #convert to float
    for i in range(0,len(a)):
        a[i]=float(a[i])
    for i in range(0,len(b)):
        b[i]=float(b[i])
    for i in range(0,len(c)):
        c[i]=float(c[i])
    for i in range(0,len(d)):
        d[i]=float(d[i])
    #new main diagonal and K
    p=[]
    p.append(b[0])
    for j in range(1,n):
        pj=b[j]-(a[j]*c[j-1])/p[j-1]
        p.append(pj)
    return p
'''
def thomas2(a,b,c,d):
    n=len(b)
    #convert to float
    for i in range(0,len(a)):
        a[i]=float(a[i])
    for i in range(0,len(b)):
        b[i]=float(b[i])
    for i in range(0,len(c)):
        c[i]=float(c[i])
    for i in range(0,len(d)):
        d[i]=float(d[i])
    #new main diagonal and K
    p=[]
    q=[]
    p.append(b[0])
    q.append(d[0])
    for j in range(1,n):
        pj=b[j]-(a[j]*c[j-1])/p[j-1] #d'
        qj=d[j]-(q[j-1]*a[j])/p[j-1] #c'
        p.append(pj)
        q.append(qj)
    x=[]
    x.append(q[n-1]/p[n-1])
    for j in range(1,n):
        xj=(q[j]-a[j]*x[j-1])/p[j]
        x.insert(0,xj)
    return x
'''

#initial values and
constants----------------------------------------------------------------------
-------------------------------------------------
E=1
Re=5000 #Reynold's number based on D between the plates
```

```
dy=.05
dt=E*Re*dy**2 #time step
a=-E/2
b=1+E
u0=0.0
ue=1
y=[0,.05,.1,.15,.2,.25,.3,.35,.4,.45,.5,.55,.6,.65,.7,.75,.8,.85,.9,.95,1.0] #y
scale for plotting
t=1
#Boundary conditions are known: u0=0, u21=1
#initial velocity profile
u=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]
#initial coefficient matrices
A=[a for i in range(1,19)]
A.insert(0,0.0)
B=[b for i in range(1,20)]
C=[a for i in range(1,19)]
C.append(0.0)
u_y=[]
u_boundary=[]

while t<241*dt:
    K=[(1-E)*u[j]+E/2*(u[j+1]+u[j-1]) for j in range(1,20)]
    #fix last K value
    K20=K[-1]-(-.5)*ue
    K.pop()
    K.append(K20)
    u=thomas(A,B,C,K)
    #add boundary conditions to velocity at next step
    u.insert(0,u0)
    u.append(ue)
    #new values of coefficients for next time step
    A=A
    C=C
    B=B
    u_boundary.append((u[10]-u[9])/u[10])
    u_y.append(u[10])
    #if t==10*dt or t==50*dt or t==60*dt or 100*dt or 240*dt:
    if t==10 or t==24 or t==50 or t==100 or t==3000:
        plt.plot(u,y)
    t+=1

'''Results-----------------------------------------------------------------------
-------------------------------------------------------------'''
#plot residual and u vs. y
plt.legend(['t=10','t=24','t=50','t=100','t=3000'])
plt.title('Velocity vs. y-Position for Various Time Steps')
plt.xlabel('u-velocity')
plt.ylabel('y-position')
plt.show()

time=np.linspace(1,241*dt,241*dt)
plt.title('Velocity Residual')
plt.plot(time,u_boundary)
plt.show()
```