```
## tout a été rédigé par Manuel Bricard, erreurs comprises

import matplotlib.pyplot as plt

## Partie 1 pas de problème, c'est du cours sur la recherche dichotomique du zéro d'une fonction
# Q1
from math import exp,tanh
from random import random,randrange

# Q2
"on doit résoudre x-tanh(x/t)=0"
def f(x,t) :
    return x-tanh(x/t)

# Q3
def dicho(f,t,a,b,eps) : # c'est un algorithme classique
    valG=a
    valD=b
    while valD-valG>eps :
        valM=(valG+valD)/2
        valeur=f(valM,t)
        if valeur==0 :
            return valM
        elif valeur*f(valG,t)<0 :
            valD=valM
        else :
            valG=valM
    return (valD+valG)/2 # return valM est aussi correct


# Q4
"""
Soit k le nombre d'itération, on a (b-a)/2**k<eps, soit k > log_2((b-a)/eps)
La complexité temporelle est en O(ln((b-a)/eps)
"""

# Q5
    # Equation (1) ou équation(2) ? Ce qui précède parle de (2), les constantes
    # ne sont pas définies...
    # La contrainte m != 0 est gérée avec l'intervalle de travail de dicho()
def construction_liste_m(t1,t2) :
    pas=(t2-t1)/499
    Lsol=[]
    if t1==0 : # pour éviter la division par 0
        t=t1+pas
    else :
        t=t1
    n=0
    while n<=500 and t<1 : # deux cas d'arrêt, 500 valeurs ou t>=1
        Lsol.append(dicho(f,t,0.001,1,10**-6))
        n+=1
        t+=pas
    Lsol+=[0]*(500-n) # on complète avec des 0 (cas t>=1) pour avoir 500 valeurs
    return Lsol
#L=construction_liste_m(0,1.4)
"""# pour vérifier que l'on obtient bien la figure de l'énoncé
plt.plot([1.4*k/499 for k in range(500)],L)
plt.show()
"""

## Partie II  base de données, pas de problème particulier sauf pour la Q8 où il faut être pointu sur
        # le fonctionnement de min()

# Q6
"""
SELECT nom FROM materiaux
WHERE t_curie<500
"""
# Q7  # une jointure
"""
SELECT nom_fournisseur, 4.5*prix_kg FROM fournisseurs
JOIN prix ON id_founisseur=id_four
WHERE id_mat=8713
"""
# Q8  #je ne sais pas si on peut faire autrement en restant dans le périmètre du programme
# la sous-requête n'est pas explicitement hors programme mais...
""" Avec sqlite, MIN ne renvoie que la première ligne vue
SELECT nom_fournisseur, 4.5*MIN(prix_kg) FROM fournisseurs
```

```
JOIN prix ON id_founisseur=id_four
WHERE id_mat=8713
"""
""" On fera donc plutôt ça pour tout avoir
SELECT nom_fournisseur, 4.5*prix_kg FROM fournisseurs
JOIN prix ON id_founisseur=id_four
WHERE id_mat=8713
AND prix_kg=(SELECT MIN(prix_kg) FROM prix WHERE id_mat=8713)
"""
# Q9
"""
SELECT nom,AVG(prix_kg) FROM materiaux
JOIN prix ON id_materiau=id_mat
GROUP BY nom
HAVING AVG(prix_kg)<50
"""

## Partie III  travail sur un damier (avec un pretexte physique)
h=200 # modifié pour le tracé de test
n=h**2

# Q10
def initialisation() :
    return [1]*n

# Q11
def initialisation_anti() :
    Lres=[]
    for i in range(h) :
        Lres.extend([(-1)**i]*h)
    return Lres

# Q12
def repliement(s) :
    return [s[k*h:(k+1)*+h] for k in range(h)]

# Q13  # là c'est un peu long pour ne pas se tromper, faire un dessin, etc...
def liste_voisins(i) :
    ligne=i//h
    col=i%h
    # la position est donnée par ligne*h+col
    return [ligne*h+(col-1)%h, ligne*h+(col+1)%h, ((ligne+1)%h)*h+col, ((ligne-1)%h)*h+col]
    # sinon on fait des tests pour savoir quand on est sur le bord et on gère la case voisine problématiqu
    # ch=case haute, cb=case bas, etc...
    """
    if ligne==0 :
        ch=n-h+col
    else :
        ch=(ligne-1)*h+col
    if ligne==h-1 :
        cb=col
    else :
        cb=(ligne+1)*h+col
    if col==0 :
        cg=(ligne+1)*h-1
    else :
        cg=i-1
    if col==h-1 :
        cd=ligne*h
    else :
        cd=i+1
    return [cg,cd,cb,ch]"""
#print(liste_voisins(0))

# Q14
def energie(s) :
    E=0
    for i in range(len(s)) : # on écrit la formule mathématique
        for j in liste_voisins(i) :
            E+=s[i]*s[j]
    return -E/2
#print(energie(initialisation_anti()))

# Q15
def test_boltzmann(delta_e,T) :
    p=exp(-delta_e/T)
    # random() renvoie un nombre entre 0 et 1, le test est donc vrai avec une proba de p
    if delta_e<=0 or random()<p :
```

```
        return True
    else :
        return False

# Q16
"""
la modification du spin en i n'a d'impact que sur le produit avec les voisins.
(Ce produit a lieu 2 fois)
La fonction calcul_delta_e1() est moins efficace puisqu'elle refait le calcul sur tout s
alors que calcul_delta_e2() ne fait que le calcul local.
"""
def calcul_delta_e2(s,i) : # pour tester les fonctions d'après
    delta_e=0
    for j in liste_voisins(i) :
        delta_e=delta_e+2*s[i]*s[j]
    return delta_e

# Q17
def monte_carlo(s,T,n_tests) :
    for k in range(n_tests) :
        ind=randrange(len(s)) # ou randrange(n)
        variation=calcul_delta_e2(s,ind)
        if test_boltzmann(variation,T) :
            s[ind]=-s[ind]

# Q18
def aimantation_moyenne(n_tests,T) :
    spins=initialisation()
    monte_carlo(spins,T,n_tests)
    somme=0
    for val in spins :
        somme+=val
    return somme/n

# Q19
"""
initialisation() est en O(n)
calcul_delta_e2() est en O(1), test_boltzmann() est en O(1)
donc monte_carlo() est en O(n_tests)
Le calcul de la moyenne est en O(n)
La complexité finale est en O(n)+O(n_test) mais je ne sais pas écrire proprement ça
sans savoir comment est n par rapport à n_test
"""
# Q20
"""
Pour prendre en compte tous les spins, on ne peut plus utliser calcul_delta_e2() qui est en O(
1)
mais calcul_delta_e1() qui utilise energie() qui est en O(n).
monte_carlo() devient en O(n*n_test), ce qui change la complexité de aimantation_moyenne()
qui devient en O(n*n_test)
"""
# Q21
"""
Lorsque la température baisse les spins de même orientation se regroupe en taches
"""


## Digression pour obtenir les images du sujet
    ## ça marche pas mal sauf pour les 2 dernières images
    ## (peut-être une erreur dans le corrigé, attention ! Ou alors je n'ai pas bien compris
    ## la physique et ai mal paramétré le mote_carlo())
"""
listT=[5,4,3,2.5,2,0.5]
c=1
for T in listT :
    plt.subplot("32"+str(c))
    c+=1
    S1=initialisation()
    monte_carlo(S1,T,10**6)
    M1=repliement(S1)
    plt.imshow(M1)
    plt.title("T="+str(T))
plt.show()
"""
## Partie IV domaine de Weiss, programmation récursive

# Q22
def explorer_voisinage(s,i,weiss,num) :
```

```
    for indV in liste_voisins(i) :
        if s[indV]==s[i] and weiss[indV]==-1 :
            weiss[indV]=num
            explorer_voisinage(s,indV,weiss,num)

# Q23
def explorer_voisinage_pile(s,i,weiss,num,pile) :
    pile=[i]
    while pile!=[] :
        indiceCourant=pile.pop()
        weiss[indiceCourant]=num
        for indV in liste_voisins(indiceCourant) :
            if s[indV]==s[indiceCourant] and weiss[indV]==-1 :
                pile.append(indV)

# Q24
def construire_domaine_weiss(s) : ## il reste une maladresse dans l'énoncé, sans conséquence
    numZone=0
    indice=0
    pile=[]
    weiss=[-1]*n
    spin=s[0]
    while indice!=n-1 :
        explorer_voisinage_pile(s,indice,weiss,numZone,pile)
        #explorer_voisinage(s,indice,weiss,numZone) # ne marche pas avec h trop grand
        while indice!=n-1 and weiss[indice]!=-1 : # on cherche une nouvelle zone (c'est à dire non explor
            indice+=1
        numZone+=1
        spin=s[indice]
    return weiss

## pour tester
import numpy as np
S=initialisation()
# construction de 4 carrés de h//6 dont les coins supérieurs gauches sont
# (h//6,h//6) (h//6,h//6+h//2) (h//6+h//2,h//6) (h//6+h//2,h//6+h//2)
v1=h//6
v2=h//6+h//2
for i in range(v1) :
    for j in range(v1) :
        S[(v1+i)*h+v1+j]=-1
        S[(v1+i)*h+v2+j]=-1
        S[(v2+i)*h+v1+j]=-1
        S[(v2+i)*h+v2+j]=-1
W=construire_domaine_weiss(S)
M=repliement(W)
plt.imshow(M)
plt.show()
#W=construire_domaine_weiss(A
```