DS2 Informatique
- CORRIGE -
Modélisation numérique d'un matériau magnétique

Partie I : Transition paramagnétique / ferromagnétique sans champ magnétique extérieur

1. Écrire les instructions nécessaires pour importer exclusivement les fonctions exponentielle (exp) et tangente hyperbolique (tanh) du module math, ainsi que les fonctions randrange et random du module random.

```
from math import exp, tanh
from random import randrange,random
```

2. À partir de l'équation $m = \tanh(m/t)$, indiquer une équation $f(x,t) = 0$, d'inconnue x que l'on doit résoudre et écrire en Python la définition de la fonction f correspondante (paramètres x et t, valeur renvoyée $f(x,t)$).

Equation à résoudre : $f(x,t) = \tanh(m/t) - m = 0$

```
def f(x,t):
    return tanh(x/t)-x
```

3. Écrire une fonction **dicho**$(\mathbf{f}, \mathbf{t}, \mathbf{a}, \mathbf{b}, \mathbf{eps})$ qui calcule une valeur approchée à *eps* près du zéro d'une fonction $f(m,t)$ de variable x et de paramètre t fixé sur un intervalle $[a,b]$. On supposera pour simplifier que la fonction dont on recherche le zéro est continue et s'annule une fois et une seule sur l'intervalle $[a,b]$.

```
def dicho(f, t, a, b, eps):
    while b - a >  eps:
                milieu = (a + b) /2
                if f(a, t) * f(milieu, t) > 0:
                        a = milieu
                else:
                        b = milieu
        return (a + b) /2
```

4. Établir l'expression de la complexité temporelle asymptotique de la fonction dicho en fonction de a, b et eps.

— L'intervalle de recherche est $[a,b]$ avec une précision *eps*. L'espace de recherche est donc de taille $b-a$ que l'on divise par 2 à chaque itération jusqu'à arriver à $eps \leqslant (b-a)/2^k$ au bout de k itérations. On obtient donc $\log eps \leqslant \log(b-a) - k\log 2$, d'où $k \leqslant \log_2 \dfrac{b-a}{eps}$ soit une complexité est en $O\left(\log \dfrac{b-a}{eps}\right)$.

5 En utilisant la fonction dicho, écrire une fonction **construction_liste_m**$(\mathbf{t1}, \mathbf{t2})$ qui construit et retourne une liste de 500 solutions de l'équation, pour t variant linéairement de t_1 à t_2 (bornes incluses). On cherchera les valeurs de m à 10^{-6} près avec un intervalle de recherche initial $m \in [0.001, 1]$.

```
def construction_liste_m(t1, t2):
        liste = []
        n = 500
        for i in range(n):
                t = t1 + (t2 - t1) / (n - 1) * i
                if t  > = 1:
                        liste.append(0)
                else:
                        liste.append(dicho(f, t, 0.001, 1 , 1e-6))
        return liste
```

Partie II : Recherche dans une base de donnéees de matériaux magnétique

6. Écrire une requête permettant d'obtenir le nom de tous les matériaux qui ont une température de Curie strictement inférieure à 500 kelvins.

```
SELECT nom FROM materiaux WHERE t_curie  <  500 ;
```

7. Écrire une requête permettant d'obtenir les noms de tous les fournisseurs proposant du nickel et le prix proposé par chacun pour 4,5 kilogrammes de nickel.

```
SELECT nom_fournisseur, prix_kg * 4.5
FROM fournisseurs
JOIN prix ON id_fournisseur = id_four
WHERE id_mat = 8713 ;
```

— Variante avec jointure des trois tables :

```
SELECT nom_fournisseur, prix_kg * 4.5
FROM fournisseurs
JOIN prix ON id_fournisseur = id_four
JOIN materiaux on id_materiaux = id_mat
WHERE nom LIKE 'nickel' ;
```

8. Modifier ou compléter la requête précédente afin d'obtenir le nom du fournisseur de nickel le moins cher ainsi que le prix à payer chez ce fournisseur pour ces 4,5 kilogrammes de nickel, En cas d'égalité du prix optimal entre plusieurs fournisseurs, on obtiendra les noms de tous les fournisseurs possibles.

```
SELECT nom_fournisseur, prix_kg * 4.5
FROM fournisseurs
JOIN prix ON id_fournisseur = id_four
WHERE id_mat = 8713
AND prix_kg = (SELECT MIN(prix_kg) FROM prix WHERE id_mat = 8713 )
```

9. Écrire une requête permettant d'obtenir le nom de tous les matériaux et le prix moyen pour un kilogramme de chacun de ces matériaux (la moyenne étant calculée pour tous les fournisseurs proposant ce matériau), en se limitant aux prix strictement inférieurs à 50 euros par kilogramme.

```sql
SELECT nom, AVERAGE(prix_kg) AS moyenne
FROM materiaux
JOIN prix ON id_materiau = id_mat
GROUP BY nom
HAVING moyenne  <  50 ;
```

Partie III : Modèle microscopique d'un matériau magnétique

10. Écrire une fonction **initialisation**() renvoyant une liste d'initialisation des domaines contenant n spins de valeur 1.

```
def initialisation():
        return [1] * n
```

11. Écrire une fonction **initialisation_anti**() renvoyant une liste s d'initialisation des domaines contenant h spins en largeur et h en hauteur en alternant les 1 et -1.

```
def initialisation_anti():
        liste=[]
        for i in range(h):
                if i\%2 == 0:
                        liste += [1] * h
                else:
                        liste += [-1] * h
        return liste
```

12. Pour afficher l'état global du matériau, il est nécessaire de convertir la liste s utilisée en un tableau de taille $h \times h$ représenté par une liste de listes. Écrire une fonction **repliement**(**s**) qui prend en argument la liste de spins s et qui renvoie une liste de h listes de taille h représentant le domaine.

```
def repliement(s):
        tableau = []
        for i in range(h):
                tableau.append(s[i*h: i*h+h])
        return tableau
```

13. Définir une fonction **liste_voisins**(**i**) qui renvoie la liste des indices des plus proches voisins du spin s d'indice i dans la liste s (dans l'ordre gauche, droite, dessous, dessus). On pourra utilement utiliser les opérations % et // de Python, qui renvoient le reste et le quotient de la division euclidienne.

```
def liste_voisins(i):
        return [(i - 1) \% h + i // h * h,   \# gauche
        (i + 1) \% h + i // h * h,   \# droite
        (i - h) \% n,                \# dessous
        (i + h) \% n,                \# dessus
        ]
```

14. Définir la fonction **energie**(**s**) qui calcule l'énergie d'une configuration s donnée ($E = -\frac{J}{2}\sum_i \sum_{j\in V_i} s_i s_j$).

```
def energie(s):
        J = 1
        E = 0
        for i in range(n):
                for v in liste_voisins(i):
                        E += s[i] * s[v]
        return -J / 2 * E
```

15. Définir une fonction**test_boltzmann**(**delta_e**, **T**) qui renvoie True si le spin change de signe, et False sinon.

```
def test_boltzmann(delta_e, T):
        if delta_e  <= 0:
                return True
        p = exp(-delta_e / T)
        return random()  <  p
```

16. Juste après avoir sélectionné au hasard l'indice i d'un spin de la liste s à basculer éventuellement, pour évaluer l'écart d'énergie delta_e entre les deux configurations avant/après, on propose deux solutions sous forme des fonctions **calcul_delta_e1** et **calcul_delta_e2**, où s[i] est le spin choisi pour être éventuellement retourné. Indiquer la solution qui vous paraît la plus efficace pour minimiser le temps de calcul en justifiant votre réponse.

— Pour calcul_delta_e1, on calcule deux fois l'énergie ce qui impose de faire à chaque fois n multiplications soit $2.n$ multiplications

— Pour calcul_delta_e2, on calcule la modification d'énergie dues aux 4 voisins ce qui nécessite 8 multiplications (coût constant). Cette méthode est donc d'autant plus rapide que n est grand.

17. En utilisant la fonction test_boltzmann, définir une fonction **monte_carlo**(**s**, **T**, **n$_t$ests**) qui applique la méthode de Monte-Carlo et qui modifie la liste s où l'on a choisi successivement n_test spins au hasard, que l'on modifie éventuellement suivant les règles indiquées dans les explications.

```
def monte_carlo(s, T, n_tests):
        for i in range(n_tests):
                spin = randrange(n)
                delta_e = calcul_delta_e2(s, spin)
                if test_boltzmann(delta_e, T):
                        s[spin] = - s[spin]
        return s
```

18. Ecrire la fonction **aimantation_moyenne**(**n_tests**, **T**) qui :

— initialise une liste des spins (avec la fonction initialisation par exemple),

— la fait évoluer en effectuant n_tests tests de Boltzmann et les inversions ventuelles qui en découlent,

— calcule et renvoie l'aimantation moyenne de la configuration à la température T (définie ici comme la somme des valeurs des spins divisée par le nombre total de spins).

```
def aimantation_moyenne(n_test, T):
        s = initialisation()
```

```
3           monte_carlo(s, T, n_tests)
4           return sum(s) / n
```

19. Évaluer la complexité asymptotique de la fonction **aimantation_moyenne**(**n_tests**, **T**) en fonction de n, nombre de spins dans le système, et de n_tests.

- aimantation_moyenne est composée de
  - initialisation qui est de complexité $O(n)$
  - monte_carlo qui est composé de plusieurs fonctions appelée n_test fois :
    - randrange que l'on suppose de complexité constante $O(1)$
    - calcul_delta_e2 qui est de complexité constante (voir question 16)
    - test_boltzmann qui est de complexité constante aussi.
  - sum qui est de complexité $O(n)$
- Donc la complexité de aimantation_moyenne est de $O(2n + n\_test)$

20. Cette complexité asymptotique serat-elle modifié si on avait voul prendre en compte toutes les interactions entre deux spins quelconques dans le système, et plus eulement entre les plus proches voisins ? Justifiez.

- La fonction calcul_delta_e2 deviendrait de complexité $O(n)$ ce qui engendredait une complexité totale $O(2n + n\_test \cdot n)$

21. Indiquer l'influence de l'augmentation de la température sur le comportement du matériau ferromagnétique.

- A basse température, on remarque que l'on réduit l'interface entre l'aimantation à 1 et à -1 car l'agitation thermique $k_B.T$ (qui tend à orienter l'aimantation dans tous les sens) baisse ce qui renforce l'effet du ferromagnétisme (qui tend à orienter toutes les aimantations dans le même sens).

Partie IV : exploration des domaines de Weiss

22 Ecrire le code de la fonction récursive **explorer_voisinage**(**s**, **i**, **weiss**, **num**) qui va :

- Pour chaque spin voisin du spin s_i, elle doit vérifier si les spins sont identiques, et s'il n'a pas déjà été affecté à un domaine de weiss précédemment.
- Dès qu'un tel spin est ajouté, on inscrit son numéro de domaine dans la liste weiss, et on explore récursivement son voisinage.

```
1 def explorer_voisinage(s, i, weiss, num):
2           weiss[i] = num \# non explicitement demandé dans l'énoncé mais indis-
  pensable
3           spin = s[i]
4           for voisin in liste_voisins(i):
5                   if weiss[voisin] == -1 and s[voisin] == spin:
6                           weiss[voisin] = num
7                           explorer_voisinage(s, voisin, weiss, num)
```

23 Ecrire le code de la fonction **explorer_voisinage_pile**(**s**, **i**, **weiss**, **num**, **pile**) qui va

- récupérer l'indice d'un spin à explorer dans la pile et le marquer dans la liste weiss,

— regarder dans son voisinage si des spins possèdent la même valeur et n'ont pas encore été affectés à un domaine, puis ajouter leurs indice dans la pile si c'est le cas.

```
def explorer_voisinage_pile(s, i, weiss, num, pile):
        pile.append(i)
        while len(pile) > 0:
                i = pile.pop()
                weiss[i] = num
                spin = s[i]
                for voisin in liste_voisins(i):
                        if weiss[voisin] == -1 and s[voisin] == spin:
                                pile.append(voisin)
```

24 Ecrire le code de la fonction **weiss** = **construire_domaine_weiss**(**s**) qui construit et renvoie la liste weiss contenant le numéro des domaines de weiss de chaque spin du domaine.

```
def construire_domaine_weiss(s):
        weiss = [-1] * n
        num = 0
        for i in range(n):
                if weiss[i] == -1:
                        pile = []
                        explorer_voisinage_pile(s, i, weiss, num, pile)
                        num += 1
        return weiss
```