

## I. TRANSITION PARAMAGNÉTIQUE/FERROMAGNÉTIQUE SANS CHAMP MAGNÉTIQUE EXTÉRIEUR

**1** Pour importer uniquement quelques fonctions à partir de modules, il faut utiliser la syntaxe

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

**2** L'équation donnée doit être résolue avec la variable  $m$  et se met sous la forme

$$f(x, t) = x - \tanh\left(\frac{x}{t}\right) = 0$$

d'inconnue  $x$ .

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

**3** On utilise la recherche par dichotomie d'une solution à l'équation  $f(x, t) = 0$  où la variable est  $x$  alors que  $t$  est un paramètre fixé. Dans cet algorithme, on conserve deux valeurs correspondant aux deux bornes de l'intervalle dans lequel se trouve la solution, initialement  $[a, b]$ . À chaque itération, on divise la largeur de l'intervalle par 2, car en appelant  $m$  le milieu de l'intervalle,

- soit  $f(a, t)$  et  $f(m, t)$  sont de signes opposés et cela signifie que la solution se trouve entre  $a$  et  $m$ , on peut donc affecter  $m$  à la variable  $b$  ;
- soit  $f(a, t)$  et  $f(m, t)$  sont de même signe et cela signifie que la solution ne se trouve pas entre  $a$  et  $m$ , on peut donc affecter  $m$  à la variable  $a$ .

On continue les itérations jusqu'à ce que la largeur de l'intervalle soit inférieure à  $2\varepsilon$  pour obtenir un résultat final égal au milieu de l'intervalle. La solution réelle est nécessairement à une distance inférieure à  $\varepsilon$  de ce résultat.

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

**4** À chaque itération, la largeur de l'intervalle de recherche est divisée par 2 à l'aide d'un nombre d'opérations toujours identique et indépendant de  $a$  et  $b$ . En nommant  $n$  le nombre d'itérations, on peut dire que l'algorithme est de complexité linéaire en  $n$ . Au bout de  $n$  itérations, la largeur de l'intervalle de recherche est  $(b - a) / 2^n$ . Il faut ainsi, pour sortir de la boucle, que

$$\frac{b - a}{2^n} \leq 2\varepsilon$$

$$\text{c'est-à-dire} \quad \frac{b - a}{\varepsilon} \leq 2^{n+1}$$

$$\text{soit} \quad n \geq \log_2\left(\frac{b - a}{\varepsilon}\right) - 1$$

donc

La complexité de la fonction `dichotomie` est en  $O\left(\log_2\left(\frac{b - a}{\varepsilon}\right)\right)$ .

**5** Il faut, pour chacune des 500 valeurs  $t$  uniformément réparties entre  $t_1$  et  $t_2$ , appliquer la fonction `dicho` si  $t < 1$  (le matériau est alors ferromagnétique), ou simplement définir une aimantation nulle si  $t \geq 1$  (le matériau est alors paramagnétique).

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

La figure 1 de l'énoncé peut être obtenue avec le code

```
t = [ 1.5*t/500 for t in range(1,501) ]
m = construction_liste_m(t[0],t[-1])
plt.plot(t,m)
plt.grid()
plt.show()
```

## II. RECHERCHE DANS UNE BASE DE DONNÉES DE MATÉRIAUX MAGNÉTIQUES

**6** Pour obtenir le nom des matériaux ayant une température de Curie inférieure à 500 kelvins, il faut exécuter la sélection

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

**7** Les noms de fournisseurs sont dans la table `fournisseurs` alors que les prix sont dans la table `prix`. Il faut donc réaliser une jointure avant la sélection, soit la requête

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

**8** Le prix minimal peut être obtenu par l'utilisation de la fonction d'agrégation `MIN` mais pas le fournisseur correspondant, car une telle fonction n'agit que sur une seule colonne du résultat. Ce n'est donc pas la projection (après l'opérateur `SELECT`) mais la sélection (après l'opération `WHERE`) qu'il faut modifier, en y incluant une condition sur le prix à l'aide d'une sous-requête, entre parenthèses.

```
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);
```

Si l'on conserve la requête de la question 7 et que l'on classe les lignes à l'aide de `ORDER BY prix`, on ne peut alors pas savoir si plusieurs fournisseurs sont aussi compétitifs. Ce n'est donc pas une réponse valable.

On parle aussi, pour ce genre de sous-requête, de « requête imbriquée » ou de « requête en cascade ». C'est un procédé très classique en SQL, qui peut s'utiliser à n'importe quel endroit de la requête principale. Une sous-requête est toujours encadrée par des parenthèses.

- 9** Il est nécessaire dans un premier temps de réaliser une jointure entre les tables `matériaux` et `prix` pour obtenir l'ensemble des matériaux. Dans un second temps, on doit grouper les lignes par matériau à l'aide de l'opérateur d'agrégation `GROUP BY`, afin d'obtenir la liste de tous les matériaux et leur prix moyen. Enfin, on sélectionne dans cette nouvelle table les matériaux dont le prix moyen est inférieur à 50.

```
SELECT nom, prix_moy FROM
  ( SELECT nom, AVG(prix_kg) AS prix_moy
    FROM matériaux JOIN prix ON id_materiau = id_mat
    GROUP BY id_mat )
WHERE prix_moy < 50;
```

Une autre possibilité existe, utilisant l'opérateur `HAVING`. Cet opérateur est équivalent à l'opérateur `WHERE`, la différence étant que `WHERE` est activé avant le processus d'agrégation alors que `HAVING` l'est après. La requête suivante donne donc aussi le résultat demandé :

```
SELECT nom, AVG(prix_kg) AS prix_moy
FROM matériaux JOIN prix ON id_materiau = id_mat
GROUP BY id_mat HAVING prix_moy < 50;
```

### III. MODÈLE MICROSCOPIQUE D'UN MATÉRIAU MAGNÉTIQUE

- 10** L'initialisation consiste en la création d'une liste de taille  $n$  remplie de 1.

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

Une autre possibilité utilisant `append` existe bien sûr, mais il est vraiment conseillé d'adopter la proposition ci-dessus.

- 11** On peut générer une liste correspondant aux deux premières lignes puis la recopier  $h/2$  fois, l'énoncé spécifiant que l'on se limite aux cas où  $h$  est pair.

```
def initialisation_anti():
    return ([1] * h + [-1] * h) * (h // 2)
```

De nombreuses réponses sont valables. Citons trois méthodes :

- la génération d'une liste de 1 où l'on modifie une ligne sur deux :

```
def initialisation_anti():
    s = [1] * n
    for i in range(1, h, 2):
        for j in range(h):
            s[i * h + j] = -1
    return s
```

- l'ajout avec `append` de 1 ou de -1 à une liste vide en fonction de la parité de la ligne

```
def initialisation_anti():
    s = []
    for i in range(n):
        if ( i // h ) % 2 == 0:
            s.append(1)
        else:
            s.append(-1)
    return s
```

- l'ajout d'une même valeur que l'on modifie à chaque ligne

```
def initialisation_anti():
    s = []
    for i in range(h):
        spin = 1 - 2 * ( i % 2 )    # 1 si i est pair
        for j in range(h):
            s.append(spin)
    return s
```

Un état réellement antiferromagnétique ne correspond pas à une simple alternance des spins sur les lignes, mais plutôt à une alternance à chaque spin, de façon analogue à un damier.

- 12** La liste obtenue précédemment doit être découpée en plusieurs listes de taille  $h$ . On utilise la syntaxe de l'extraction.

```
def repliement(s):
    L = []
    for i in range(h):
        L.append(s[h * i : h * (i + 1)])
    return L
```

- 13** Les voisins du spin  $s$  considéré sont le plus souvent les cases adjacentes. Une fois ces voisins déterminés, il faut vérifier que le spin  $s$  n'est pas au bord de l'échantillon, et éventuellement modifier les voisins concernés.

```
def liste_voisins(i):
    # Cas général
    gauche = i - 1
    droite = i + 1
    bas = i + h
    haut = i - h
    # Modifications en cas de présence sur un bord
    if i % h == 0:
        gauche = i - 1 + h
    if (i + 1) % h == 0:
        droite = i + 1 - h
    if i >= (h - 1) * h:
        bas = i % h
    if i < h:
        haut = ( i - h ) % n
    return [gauche, droite, bas, haut]
```

De nombreuses autres idées sont possibles. Par exemple, on remarque que dans la division euclidienne de la position  $i$  du spin par  $h$ , le quotient est le nombre de lignes au-dessus du spin et le reste représente la position du spin sur sa ligne. On obtient ainsi le voisin de gauche en ajoutant  $h$  fois le quotient de  $i$  par  $h$  et le reste de  $i - 1$  par  $h$  qui vaut  $h - 1$  si  $i$  est multiple de  $h$ . On peut faire de même pour les autres voisins de  $i$ . Cette technique est plus rapide à calculer puisqu'elle évite l'utilisation des tests.

```
def liste_voisins(i):
    gauche = (i // h) * h + (i - 1) % h
    droite = (i // h) * h + (i + 1) % h
    bas = (i + h) % n
    haut = (i - h) % n
    return [gauche, droite, bas, haut]
```

**14** On utilise la fonction `liste_voisins`. L'équation 3 de l'énoncé précise que l'énergie est la somme pour chaque spin des produits de spins avec ses voisins, le tout étant multiplié par  $-J/2$  avec  $J = 1$ .

```
def energie(s):
    e = 0
    for i in range(n):
        for j in liste_voisins(i):
            e = e + s[i] * s[j]
    return - e / 2
```

**15** Le spin doit changer de signe dans deux cas :

- si  $\Delta E$  est négatif ;
- si une valeur aléatoire choisie entre 0 et 1 est inférieure à  $p$ .

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

Il est toujours préférable, dans les fonctions qui retournent des valeurs booléennes, d'éviter d'utiliser `if` sous la forme

```
if condition_booleenne:
    return True
else:
    return False
```

qui est équivalent à un simple `return condition_booleenne`.

On pourrait se dire que le premier test est inutile, puisque le deuxième sera toujours vrai si  $\Delta E$  est négatif. Mais ce premier test permet d'accélérer significativement le traitement des cas où  $\Delta E$  est négatif.

**16** Dans la fonction `calcul_delta_e1`, une copie de la liste `s` est réalisée, le spin à basculer `y` est modifié et la fonction `energie` est appliquée sur `s` et sa copie. Cela demande un nombre de calculs proportionnel à  $n$ .

Au contraire, dans la fonction `calcul_delta_e2`, seuls les 4 calculs des échanges avec les spins voisins du spin à basculer sont réalisés, puisque tous les autres produits constituant l'énergie globale de l'échantillon restent inchangés. Cela demande un nombre de calculs borné par une constante.

La fonction `calcul_delta_e2` est la plus efficace.

Pour faire le calcul menant à l'écriture de la fonction `calcul_delta_e2`, posons  $s_i$  le spin à basculer et  $s_1$  à  $s_4$  les spins voisins. Alors,

$$E_{\text{avant}} = E_0 - \frac{1}{2} s_i s_1 - \frac{1}{2} s_i s_2 - \frac{1}{2} s_i s_3 - \frac{1}{2} s_i s_4 \\ - \frac{1}{2} s_1 s_i - \frac{1}{2} s_2 s_i - \frac{1}{2} s_3 s_i - \frac{1}{2} s_4 s_i$$

et

$$E_{\text{après}} = E_0 + \frac{1}{2} s_i s_1 + \frac{1}{2} s_i s_2 + \frac{1}{2} s_i s_3 + \frac{1}{2} s_i s_4 \\ + \frac{1}{2} s_1 s_i + \frac{1}{2} s_2 s_i + \frac{1}{2} s_3 s_i + \frac{1}{2} s_4 s_i$$

soit  $E_{\text{après}} - E_{\text{avant}} = 2 s_i s_1 + 2 s_i s_2 + 2 s_i s_3 + 2 s_i s_4$

**17** La fonction `monte_carlo` doit, `n_tests` fois, choisir aléatoirement un des  $n$  spins et le modifier si le test de Boltzmann est vérifié.

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

**18** La fonction `aimantation_moyenne` exécute la méthode de `monte_carlo` puis calcule la moyenne des spins.

```
def aimantation_moyenne(n_tests, T):
    s = initialisation()
    monte_carlo(s, T, n_tests)
    somme = 0
    for spin in s:
        somme = somme + spin
    return somme / n
```

On peut ainsi générer une courbe simulée proche de la courbe théorique représentée sur la figure 1 de l'énoncé, à l'aide du code

```
n_tests = n * 100
aimantation = []
N = 20 # nombre de points de la figure
temperatures = [ T*5/N for T in range(1,N+1) ]
for T in temperatures:
    aimantation.append(aimantation_moyenne(n_tests,T))
plt.plot(temperatures,aimantation)
plt.grid()
plt.show()
```

**19** La complexité de la fonction `initialisation` est linéaire en fonction de  $n$ , tandis que celle de la fonction `monte_carlo` est linéaire en fonction de  $n_{\text{tests}}$  car la fonction `test_boltzmann`, comme `calcul_delta_e2` qu'elle appelle, est de complexité constante.

La complexité de la fonction `aimantation_moyenne` est donc en  $O(n + n_{\text{tests}})$ .

**20** Si on souhaite prendre en compte toutes les interactions entre les spins de l'échantillon, et en supposant que l'équation (3) devient

$$E = -\frac{J}{2} \sum_i \sum_{j \neq i} s_i s_j$$

alors, en notant  $S$  l'aimantation totale de l'échantillon, la variation d'énergie s'écrit

$$\Delta E = J s_i \sum_{j \neq i} s_j = J s_i (S - s_i)$$

En supposant que l'on calcule une première fois  $S$  à l'initialisation et qu'on la maintient à jour entre deux tests de Boltzmann, le calcul de  $\Delta E$  reste à complexité constante comme avec la fonction `calcul_delta_e2`.

La complexité de `aimantation_moyenne` resterait en  $O(n + n_{\text{tests}})$ .

Si on considère un coefficient  $J$  différent pour chaque couple de spins, alors le calcul de  $\Delta E$  nécessite cette fois d'utiliser une forme modifiée de la fonction `calcul_delta_e1`, de complexité en  $O(n)$ . La complexité de `aimantation_moyenne` deviendrait  $O(n \cdot n_{\text{tests}})$ .

**21** On observe sur la figure 5 que, pour des **températures supérieures à la température de Curie**, il n'y a **pas de formation de domaines magnétiques**. Les spins restent avec une valeur aléatoire, l'aimantation moyenne est nulle, ce qui correspond au **comportement paramagnétique** du matériau.

Au contraire, lorsque la **température est inférieure à la température de Curie**, des **domaines magnétiques se forment** et localement, l'aimantation peut devenir unitaire. Cela correspond au **comportement ferromagnétique** du matériau.

#### IV. EXPLORATION DES DOMAINES DE WEISS

**22** La fonction `explorer_voisinage` étant récursive, elle a parmi ses arguments une liste de valeurs `weiss` déjà initialisée et partiellement traitée, qui est modifiée à chaque exécution. La fonction `explorer_voisinage` se contente donc, pour chaque pixel voisin du pixel  $i$  non encore traité, d'y noter le numéro de domaine et d'exécuter à nouveau la fonction sur ce pixel, à condition qu'il soit bien égal au pixel  $i$ .

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

**23** Dans cette fonction, une pile est utilisée. On y ajoute à chaque pixel  $i$  traité les pixels voisins non encore traités et identiques à  $i$ . À chaque itération de l'algorithme, on traite le pixel qui se trouve en haut de la pile en l'en enlevant et en marquant sa couleur dans la liste `weiss`. Il est important d'initialiser la pile en y mettant le premier pixel à traiter. La boucle s'arrête lorsque la pile est vide.

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

**24** La fonction `construire_domaines_weiss` doit initialiser les variables `num` et `weiss`, puis exécuter `explorer_voisinage_pile` pour chaque pixel non encore traité. Une boucle `for` sur les valeurs de `i` permet de s'assurer que tous les pixels seront bien traités une fois.

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

Pour réaliser la figure présentée par l'énoncé, il faut

- initialiser un échantillon ;
- lui appliquer la méthode de Monte-Carlo pour faire évoluer les spins ;
- marquer chaque domaine de Weiss ;
- afficher l'image.

Pour ce faire, on peut exécuter le code

```
ntests = n * 100
T = 0.8
s = initialisation() # ou initialisation_anti()
monte_carlo(s,T,ntests)
w = construire_domaines_weiss(s)
plt.imshow(repliement(w), cmap = 'gray')
plt.show()
```

En pratique, on remarque pour des températures faibles que l'algorithme de Monte-Carlo fait difficilement évoluer l'échantillon à partir d'une configuration ferromagnétique correspondant à une énergie négative déjà globalement minimale. Il est alors possible de prendre une initialisation antiferromagnétique d'énergie nulle, qui permet d'obtenir plus rapidement une figure proche des figures 6 et 7 de l'énoncé, par exemple avec une température autour de 0,8 et environ un million de tests.