Corrigé du DS ITC n°1 Mines 2022

Partie I

1. ```
   from math import exp, tanh
   from random import randrange, random
   ```

   On peut aussi utiliser la syntaxe :

   `import math as m` et `import random as rd` et les fonctions de ces bibliothèques seront ensuite accessibles par `m.exp`, `m.tanh`, `rd.randrange` et `rd.random`

2. ```
   def f(x,t) :
       return x - tanh(x/t)
   ```

3. ```
   def dicho(f,t,a,b,eps) :
       while b-a > eps :
           x = (a+b)/2
           if f(x,t) == 0 : return x
           elif f(a,t)*f(x,t) > 0 :
               a == x
           else :
               b = x
       return (a+b)/2
   ```

4. Soit p le nombre de tours de boucle `while` nécessaires, avec à chaque tour une division par 2 de l'intervalle. L'algorithme se termine lorsque l'intervalle atteint une largeur de `eps`, ce qui donne (en ordre de grandeur) :

$$\frac{b-a}{2^p} = \text{eps} \implies \boxed{p = \log_2\left(\frac{b-a}{\text{eps}}\right)}$$

   L'algorithme est en $O(p)$.

5. ```
   def construction_liste_m(t1,t2) :
       solutions = []
       pas = (t2 - t1)/499        # avoir les deux bornes incluses
       for i in range(500) :
           t = t1 + pas*i
           if t >= 1 :
               m = 0
           else :
               m = dicho(f,t,0.001,1,1e-6)
           solutions.append(m)
       return solution
   ```

Partie III

10. On peut proposer :

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

11. Comme il est nécessaire de pouvoir passer facilement de l'indice i d'un spin dans la simple liste à son indice de ligne l et son indice de colonne c dans la représentation matricielle des Figures 2 et 3, on commence par écrire une fonction auxiliaire qui calcule l et c à partir de i : la formule utilisée est $i = h \times l + c$, avec $0 \leqslant c \leqslant h - 1$. Le bon outil est celui de la division euclidienne des entiers.

```
def ligne_colonne(i) :
  l = i // h
  c = i % h
  return (l,c)
```

Il est alors plus facile d'écrire la fonction demandée :

```
def initialisation_anti() :
  L = []
  for i in range(n) :
    l,c = ligne_colonne(i)
    if l % 2 == 0 :  # si l est paire
      L.append(1)
    else :
      L.append(-1)
  return L
```

12. On va utiliser la fonction `ligne_colonne` écrite à la question précédente :

```
def repliement(s) :
  n = len(s)
  M = [ h*[0] for j in range(h) ]
  for i in range(n) :
    l,c = ligne_colonne(i)
    M[l][c] = s[i]
  return M
```

13. Encore une fois la fonction `ligne_colonne` va servir à écrire la fonction suivante :

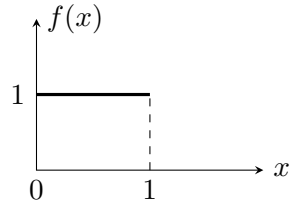
```
def liste_voisins(i) :
  l,c = ligne_colonne(i)
  linf = l - 1
  if linf < 0 : linf += h
  lsup = l + 1
  if lsup >= h : lsup += - h
  cinf = c - 1
  if cinf < 0 : cinf += h
  csup = c + 1
  if csup >= h : csup += - h
  L = [ s[l*h+cinf], s[l*h+csup], s[linf*h+c], s[lsup*h+c] ]
  return L
```

14. On peut proposer :

```
def energie(s) :
    n = len(s)
    valeur = 0
    for i in range(n) :
        L = liste_voisins(i)
        somme_spins_voisins = 0
        for j in L :
            somme_spins_voisins += L[j]
        valeur = s[i] * somme_spins_voisins
    return - 0.5 * valeur
```

15. Lorsque $\Delta E > 0$, il faut *simuler une loi de Bernouilli* : l'univers est composé de deux évènements $\Omega = \{\text{True}, \text{False}\}$ avec des probabilités d'occurence respectives p et $1-p$.

Pour y arriver, on se sert de la loi uniforme de support $[0,1]$, dont la densité de probabilité est donnée ci-dessous :



$$f(x) = \begin{cases} 1 & \text{si} \quad x \in [0,1] \\ 0 & \quad\quad \text{ailleurs} \end{cases}$$

Soit X une variable aléatoire réelle vérifiant cette loi : l'ensemble de ses réalisations est $\mathbb{R}$ et, par définition, la probabilité d'obtenir une valeur x vérifiant $0 \leqslant x \leqslant p$, avec $p \leqslant 1$ est donnée par :

$$\mathscr{P}(0 \leqslant x \leqslant p) = \int_0^p f(x)\,\mathrm{d}x = \int_0^p \mathrm{d}x = p$$

Il s'ensuit que pour réaliser la simulation de la loi de Bernouilli en langage Python , on réalise donc un tirage au sort d'une valeur réelle x avec la fonction `random.random()` qui simule une loi uniforme de support [0,1]. Si la valeur de x obtenue est :

- inférieure ou égale à p, on renvoie `True` ;
- strictement supérieure à p, on renvoie `False`

```
def test_boltzmann(delta_e,T) :
    if delta_e <= 0 :
        return True
    else :
        p = np.exp(-delta_e/T)
        x = random()    # la fonction a été importée avec le nom random
        if x <= p :
            return True
        else :
            return False
```

16. La solution 2 est évidemment meilleure car elle ne réalise pas de copie de liste (opération de complexité temporelle n) et n'appelle pas `energie(s)` qui est de complexité temporelle n elle aussi.

17. Comme l'énoncé n'indique rien sur les `n_tests` spins, on ne supposera pas qu'ils sont distincts deux à deux.

```
def monte_carlo(s, T, n_tests) :
  for k in range(n_tests) :
    i = randrange(0,n)    # tirage au sort d'un indice entre 0 et n-1
    if test_boltzmann(calcul_delta_e2(s,i), T) :
      s[i] *= -1    # on change le signe du spin
```

Remarque :

La fonction s'appelle une *procédure* car elle ne renvoie rien (pas de `return`). La liste `s` passée en paramètre est modifiée par la fonction : à la fin de son exécution, `s` ne sera plus identique à la liste passée en paramètre.

18. On peut proposer :

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

19. La fonction `initialisation` possède une complexité linéaire en $O(\texttt{n})$. La fonction `monte_carlo` est en $O(\texttt{n\_tests})$ grâce à `calcul_delta_e2` qui est s'exécute à temps constant. Enfin, le calcul de l'aimantation moyenne est de complexité linéaire $O(\texttt{n})$.

Ainsi, la fonction `aimantation_moyenne` est en $O(\texttt{n\_tests + n}) = O(\texttt{n})$ puisque `n_tests` $\leqslant n$.

20. Dans le cas où l'on prendrait en compte toutes les interactions, le calcul de ΔE serait de complexité $O(\texttt{n})$. Comme il est répété `n_tests` fois dans la boucle `for` de `monte_carlo`, cela change donc la complexité asymptotique en $O(\texttt{n\_tests x n})$.

21. L'augmentation de la température a tendance à rendre aléatoire la répartition des domaines $+1$ et -1, et donc à supprimer l'aimantation du matériau.

22. On peut proposer :

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

23. La fonction est maintenant écrite de façon itérative, avec une structure explicite de pile. On peut proposer :

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

24. On peut écrire :

```
def construire_domaines_weiss(s) :
  weiss = n*[-1]
  num,i = 0,0
  while -1 in weiss :
    while weiss[i] != -1 :     # on cherche un nouveau point de départ
      i += 1
    pile = []
    explorer_voisinage(s,i,weiss,num,pile)
    num += 1
  return weiss
```