Représentation des nombres
Approximation des calculs flottants

L'essentiel du cours

Codage des entiers naturels

Les nombres entiers naturels sont gérés à travers leur décomposition binaire.
Les syntaxes `0b`..., `0o`... ou `0x`... convertissent en base 10 les nombres dont les décompositions sont en pointillés respectivement en base 2, en base 8 et en base 16.
Il est à noter qu'un bit peut coder deux informations : 0 ou 1 alors qu'un octet peut coder $2^8 = 256$ informations. Classiquement, on peut coder sur un octet tous les entiers $k \in [\![0, 255]\!]$.

Un ordinateur disposant d'un processeur à 32 bits peut coder théoriquement tous les entiers entre 0 et $2^{32} - 1$. Cependant, il reste la question de coder les entiers négatifs par exemple.

Codage des entiers négatifs

Bit de signe

Une première façon de coder sur 32 bits par exemple serait de réserver un bit – le plus à gauche par exemple – pour coder le signe, codage à 0 pour le signe positif et codage à 1 pour le signe négatif. Les 31 bits restants permettent alors de coder l'entier positif associé à la valeur absolue du nombre de départ.
Ainsi, on pourrait coder tous les entiers relatifs de valeur absolue comprise entre 0 et $2^{31} - 1$. Le problème avec cette représentation est que les opérations d'addition ou de soustraction demandent des efforts, en tout cas ne sont pas immédiates à partir des opérations connues sur les nombres binaires.

Complément à 1

Un autre façon de coder les entiers relatifs sur n bits est, étant donné un entier naturel $x \in [\![0, 2^{n-1} - 1]\!]$, de coder l'opposé de x en codant l'entier $y = 2^n - 1 - x \in [\![2^{n-1}, 2^n - 1]\!]$.
Étant donné un entier naturel x de représentation binaire $[a_{n-1}, a_{n-2}, \cdots, a_0]$, l'opposé de x sera alors codé :

$$[1 - a_{n-1}, 1 - a_{n-2}, \cdots, 1 - a_0].$$

Cela revient à modifier chaque bit dans le n-uplet.

Le problème est que 0 et -0 correspondent à deux représentations différentes.

Complément à 2

Encore une autre façon de coder les entiers relatifs sur n bits est, étant donné un entier naturel $x \in [\![0, 2^{n-1} - 1]\!]$, de calculer sa décomposition binaire $[a_{n-1}, \cdots, a_1, a_0]$.
On décide alors de coder l'opposé de x en codant l'entier naturel $y = 2^n - x$.
Comme l'entier x est strictement inférieur à 2^{n-1}, alors l'entier y est supérieur ou égal à 2^{n-1}, donc le bit de poids fort – c'est-à-dire le bit le plus à gauche – vaut 1 et ensuite, on obtient la décomposition binaire du complément à 1 de l'entier x. On peut ainsi coder tous les entiers x entre -2^{n-1} et $2^{n-1} - 1$. L'entier $2^{n-1} - 1$ sera codé sur n bits selon $[0, 1, \cdots, 1]$ et l'entier -2^{n-1} sera codé sur n bits selon $[1, 0, \cdots, 0]$.

Pour calculer le complément à 2 d'un entier x entre 1 et 2^{n-1} – autrement dit, pour coder l'entier relatif $-x$, on se place pour l'instant dans le cas particulier où x est différent de 2^{n-1}. Dans ce cas,

- $\longrightarrow$ on décompose x en binaire, sur n bits ; le bit de poids le plus fort vaut pour l'instant 0 ; la décomposition ne comporte pas que des 0
- $\longrightarrow$ on inverse tous les chiffres dans cette décomposition ; le bit de poids le plus fort vaut maintenant 1 ; la décomposition ne comporte pas que des 1
- $\longrightarrow$ cette nouvelle décomposition correspond à la décomposition binaire d'un entier k entre 2^{n-1} et $2^n - 2$
- $\longrightarrow$ on effectue la décomposition binaire de l'entier $k + 1$
- $\longrightarrow$ cette nouvelle décomposition binaire correspond au codage de l'entier relatif $(-x)$.

Dans le cas où l'on calcule le complément à 2 de 2^{n-1}, on suit la même démarche :

- $\longrightarrow$ décomposition binaire sur n bits de 2^{n-1} : $[1, 0, \cdots, 0]$ avec $(n-1)$ zéros
- $\longrightarrow$ inversion des bits : $[0, 1, \cdots, 1]$ avec $(n-1)$ chiffres 1
- $\longrightarrow$ ajout de 1 : $[1, 0, \cdots, 0]$

On dispose d'une autre méthode pour coder $(-x)$, dans le cas où x est un entier entre 1 et $2^{n-1} - 1$.
On décompose l'entier x en binaire. Dans cette décomposition, on lit la décomposition de droite à gauche. Tant que l'on n'a pas rencontré le premier chiffre 1, on garde les chiffres. À partir du bit juste à gauche du premier 1, on inverse les chiffres. Cette méthode marche également si $x = 2^{n-1}$. Dans ce cas, le premier 1 lu à partir de la droite est le seul 1 tout à gauche. Il n'y a alors pas lieu de modifier quoi que ce soit à la gauche (vide) du seul 1 lu.

Un exemple

On veut coder -833 sur $n = 16$ bits

Voici le détail de calcul du codage selon la première méthode :

- on code en binaire l'entier 833 :

$$[0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,1]$$

- on inverse tous les chiffres binaires :

$$[1,1,1,1,1,1,0,0,1,0,1,1,1,1,1,0]$$

- on ajoute 1 à l'entier dont la décomposition binaire est celle ci-dessus :

$$[1,1,1,1,1,1,0,0,1,0,1,1,1,1,1,1]$$

  Il s'agit du codage de l'entier -833.

Voici le détail du calcul du codage selon la deuxième méthode :

- on code en binaire l'entier 833 :

$$[0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,1]$$

- on lit de droite à gauche et on inverse à gauche du premier bit valant 1 :

$$[1,1,1,1,1,1,0,0,1,0,1,1,1,1,1,1]$$

  Il s'agit du codage de l'entier -833.

Compatibilité avec l'addition

On montre facilement que si x est un entier entre 0 et $2^{n-1}-1$ et si $\overline{y}$ est l'entier complément à l'entier y, alors $x+\overline{y}$ est toujours congru à $x-y$ modulo 2^n. Il suffit alors pour effectuer les additions/soustractions de ne retenir que les $n-1$ bits de droite dans les décompositions binaires.
En effet, ces opérations d'addition/soustractions s'effectuent modulo 2^n, puis on ne retient que les n bits les plus à gauche en négligeant le dépassement.
Plus précisément, si x est un entier entre 1 et 2^{n-1}, on pose $[a_{n-1},\cdots,a_0]$ sa décomposition binaire sur n bits.
L'inversion des bits a_i en $1-a_i$ fournit la décomposition binaire de l'entier :

$$\begin{aligned} t &= \sum_{i=0}^{n-1}(1-a_i)\cdot 2^i \\ &= \sum_{i=0}^{n-1}2^i - \sum_{i=0}^{n-1}a_i\cdot 2^i \\ &= 2^n-1-x. \end{aligned}$$

Ensuite, on ajoute 1 à cette quantité avant de reprendre sa décomposition binaire : on tombe sur 2^n-x.
On en déduit que si x et y sont deux entiers entre -2^{n-1} et $2^{n-1}-1$, pour calculer la somme $x+y$, en notant C_x et C_y les codages des entiers x et y sur n bits :

- soit x et y sont des entiers naturels, auquel cas, les codages C_x et C_y correspondent aux décompositions binaires de x et de y : il suffit de faire l'addition en binaire de ces deux décompositions
- soit $x < 0$ et $y \geqslant 0$, auquel cas, le codage C_x est la décomposition binaire sur n bits de $2^n + x$ et C_y est la décomposition binaire de y. Si l'on effectue l'addition binaire sur C_x et C_y, on obtient la décomposition binaire de $2^n + x + y$ sur $n + 1$ bits. Soit $x+y \geqslant 0$, soit $x+y < 0$, mais quoi qu'il arrive, la décomposition binaire obtenue pour $2^n + x + y$ permet en ne retenant que les n bits les plus à droite (on élimine le dépassement) de retrouver $x + y$ ou le codage de $x + y$ par son complément à 2.
- les deux autres cas sont identiques en travaillant modulo 2^n.

Dépassement à 2 et type int

Si l'on veut additionner deux entiers x et y entre -2^{n-1} et $2^{n-1} - 1$, il peut arriver que la somme sorte de cet intervalle d'entiers. En particulier, ce problème se pose systématiquement lorsque $x > 2^{n-2}$ et $y > 2^{n-2}$. On obtient alors un dépassement de capacité. La version Python 3 ne fait pas la distinction entre les entiers qui dépassent ou non cette limite. Dans d'autres versions, le type long apparaît au lieu du type int.

Implémentations

Première fonction de codage du complément à deux

Voici une fonction qui code en binaire tous les entiers entre -2^{n-1} et $2^{n-1} - 1$ sur n bits, ce codage s'appuyant sur la décomposition binaire de $2^n - 1 - x$ pour coder $-x$:

```
def Bin(x) :
    if x==0 :
        return []
    else :
        return Bin(x//2)+[x%2]

def Calcul(L) :
    s=0
    p=1
    for i in range(len(L)) :
        s+=p*L[-i-1]
        p*=2
    return s

def Codage1(n,x) :
    if x>=0 :
        L=Bin(x)
        return (n-len(L))*[0]+L
    else :
        L=Codage1(n,-x)
        M=[1-k for k in L]
        return Bin(Calcul(M)+1)
```

Deuxième fonction de codage du complément à deux

Cette deuxième fonction porte sur l'inversion des bits à gauche du premier 1.

```
def Codage2(n,x) :
    if x>=0 :
        L=Bin(x)
        return (n-len(L))*[0]+L
    else :
        L=Codage2(n,-x)
        i=0
        while L[-i-1]==0 :
            i+=1
        return [1-k for k in L[:-i-1]]+L[-i-1:]
```

Représentation des réels

Décomposition diadique

Tout nombre réel de l'intervalle $[0, 1[$ se décompose sous la forme :

$$x = \sum_{n=1}^{+\infty} \frac{b_n}{2^n},$$

où pour tout $n \in \mathbb{N}^*$, $b_n \in \{0, 1\}$. On écrit alors :

$$(x)_2 = 0, b_1 b_2 \cdots$$

Cette décomposition n'est pas unique car par exemple :

$$\frac{1}{2} = 0, 100 \cdots = 0, 0111 \cdots$$

Pour obtenir la décomposition diadique d'un réel entre 0 et 1, on opère par multiplications successives par 2 et on ne retient que les parties entières.

Décomposition des flottants

Tout nombre réel peut se décomposer selon :

$$x = n + \alpha,$$

où n est la partie entière de x : $n = \lfloor x \rfloor$ et α est sa partie décimale : $\alpha \in [0, 1[$.
L'ordinateur peut donc coder sur autant de bits que nécessaire l'entier n, puis il peut coder les premiers termes dans le développement diadique du réel α.
Tous les nombres réels x qui ne sont pas de la forme $\frac{k}{2^n}$, où $k \in \mathbb{Z}$ et $n \in \mathbb{N}$ admettront une partie décimale associée à un développement diadique infini – c'est-à-dire comportant un nombre infini de termes non nuls. L'ordinateur ne peut gérer cette quantité infinie d'informations et doit donc procéder à une approximation des flottants, le cas échéant.

Type float

La langage de programmation Python fait la différence entre le type entier int et le type flottant float. Par exemple, type(1) renvoie int, alors que type(1.0) renvoie float.
Les flottants sont représentés sous la forme d'une mantisse binaire ou décimale – c'est-à-dire le nombre appartenant à $[1, 2[$ obtenu en calculant la décomposition diadique ou décimale de ce nombre puis en décalant la virgule pour ne faire apparaître qu'un seul chiffre à gauche de la virgule le cas échéant – et d'un exposant. En pratique, comme la mantisse m est dans $[1, 2[$, on ne code que le nombre $m - 1 \in [0, 1[$, ce qui est plus pratique car on commence à coder uniquement les décimales après la virgule et pas le premier chiffre (partie entière nulle).

Ainsi, un nombre flottant décimal s'écrit de la forme x=me+N ou x=me-N, où m est la mantisse en base 10 et l'entier N indique le nombre de décalages de la virgule effectués, le signe + indiquant des décalages vers la droite de cette virgule et le signe - indiquant des décalages de cette virgule vers la gauche.
Là encore, le système limite les dépassements possibles. Le fait d'occasionner un dépassement de capacité comme c'est le cas par exemple si l'on veut calculer une grande puissance de π génère une erreur OverFlowError. L'enjeu dans des calculs volumineux est de simplifier au maximum les calculs avant de passer à l'itération suivante, éventuellement de mettre une partie du calcul en mémoire...

Erreurs d'approximations

Tout calcul sur les flottants va de fait générer des erreurs d'approximations qui peuvent conduire à la longue à des amplifications de ces erreurs pour finalement conduire à des résultats aberrants. Ces erreurs apparaissent essentiellement lors des modélisations numériques : résolutions approchées de solutions à une équation, itération de suites ou de processus dynamiques de manière générale.

Implémentations selon la norme IEEE-754

Types float32 ou float64

Selon le type float32 (simple précision) un nombre réel sera codé sur 32 bits, avec la répartition suivante :

- un bit pour le signe : 0 pour + et 1 pour −
- un octet pour l'exposant ; cet exposant e peut donc théoriquement parcourir la plage $[\![-128, 127]\!]$ mais en pratique la plage $[\![-126, 127]\!]$, deux entiers étant associés au codage de 0 ou de ∞, ce qui est largement suffisant pour décrire les quantités utiles en physique par exemple ; l'exposant E est en fait remplacé par $E' = E + 2^7 - 1$ avant d'être codé
- 23 bits pour la mantisse, le nombre avant la virgule en binaire valant 1 ; on peut donc coder des nombres $m \in [1, 2[$ tels que :

$$m = 1 + \sum_{k=1}^{23} \frac{\varepsilon_k}{2^k}.$$

On obtient ainsi un 32-uplet où les données sont dans l'ordre de lecture : le signe, puis l'exposant pour les huit bits suivants, puis enfin la mantisse pour les 23 bits finaux.

Selon le type float64 (double précision) un nombre réel sera codé sur 64 bits, avec la répartition suivante :

- un bit pour le signe : 0 pour + et 1 pour −

- onze bits pour l'exposant ; cet exposant peut donc théoriquement parcourir la plage $[\![-1024, 1023]\!]$ mais en pratique la plage $[\![-1022, 1023]\!]$; l'exposant E est en fait remplacé par $E' = E + 2^{10} - 1$ avant d'être codé
- 52 bits pour la mantisse, le nombre avant la virgule en binaire valant 1 ; on peut donc coder des nombres $m \in [1, 2[$ tels que :

$$m = 1 + \sum_{k=1}^{52} \frac{\varepsilon_k}{2^k}.$$

On obtient ici un 64-uplet où les données sont dans l'ordre de lecture : le signe, puis l'exposant pour les onze bits suivants, puis enfin la mantisse pour les 52 bits finaux.

Codage en 32 bits des flottants

Voici le codage du signe :

```
def Signe(x) :
    if x>=0 :
        return 0
    else :
        return 1
```

Voici le codage de l'exposant :

```
def Bin(x) :
    if x==0 :
        return []
    else :
        return Bin(x//2)+[x%2]

def Exposant(e) :
    L=Bin(e+127)
    return (8-len(L))*[0]+L
```

Voici le codage de la mantisse :

```
def Diadique(x) :
    L=[]
    i=0
    x=x-1
    while i!=23 and x!=0 :
        y=2*x
        L.append(int(y//1))
        x=y%1
        i+=1
    return L+(23-len(L))*[0]
```

Exemple pratique de codage

On veut par exemple coder en float32 le nombre :

$$x = 833 + \frac{1}{833}.$$

Premièrement, le signe de x est positif : le bit de signe vaut 0.
Ensuite, il faut mettre x sous la forme :

$$x = m \cdot 2^e,$$

avec e un entier et $m \in [1, 2[$. On trouve :

$$x = \left(\frac{x}{2^9}\right) \cdot 2^9.$$

Pour coder l'exposant $e = 9$, on ajoute 127, ce qui donne 136 et on décompose en binaire sur un octet ce qui donne :

$$[1, 0, 0, 0, 1, 0, 0, 0]$$

Pour coder la mantisse $m = \frac{x}{2^9}$, on multiplie par 2 et on prend la partie entière, pour avoir 23 décimales après la virgule. On obtient :

$$[1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1]$$

Voici la représentation du réel x en flottant simple précision :

$$[0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1].$$

Exercice 1

1. Rédiger une fonction itérative Bin(x) qui à partir d'un entier naturel x renvoie sous forme de liste sa décomposition binaire. Par exemple, Bin(6) doit renvoyer [1,1,0].
2. Rédiger une fonction Dec(L) qui à partir d'une liste L composée de 0 et de 1 renvoie le seul entier naturel dont la décomposition binaire est associée à la liste L.
3. Rédiger une fonction Complement2(x,n) qui calcule la décomposition binaire codant le complément à deux de l'entier x sur n bits. On s'attend à trouver par exemple Complement2(5,4) $\longrightarrow$ [1,0,1,1].
4. Rédiger une fonction Oppose2(x,n) donnant la valeur de l'entier dont la décomposition binaire est donnée par la fonction précédente.
5. Tester les calculs Oppose2(x,n)+y et $x-y$ modulo 2^n, pour $n=4$ et x et y variant dans $[\![0,15]\!]$.
6. Quels sont les entiers dont les codages sur un octet sont $[0,0,0,1,0,1,1,1]$ et $[1,0,0,0,1,1,0,0]$?

Exercice 2

1. Rédiger un fonction permettant le calcul de la décomposition diadique d'un réel x entre 0 et 1 de la forme $x=\dfrac{k}{2^n}$, où k et n sont deux entiers naturels.
2. Quelles sont les décompositions diadiques de $\dfrac{1}{3}$? de $\dfrac{1}{10}$?
3. Comment expliquer le résultat obtenu par la syntaxe : print(1-0.2-0.2-0.2-0.2-0.2) ?

Exercice 3

1. (a) Rédiger une fonction permettant de stocker les n premières valeurs de la suite récurrente :
$$\begin{cases} u_0 = 2\pi \\ \forall n \in \mathbb{N}\ u_{n+1} = 2\pi\ (\cos u_n + \sin u_n) \end{cases}.$$
   (b) Comment expliquer le résultat fourni pour $n=50$?
2. Faire afficher pour ε variant dans $\left\{\dfrac{1}{10^i}\ ;\ 2\leqslant i\leqslant 17\right\}$ les valeurs d'approximation de la dérivée de la fonction $f : x\longmapsto x^2$ en 1 par :
$$f'(1)\simeq\frac{f(1+\varepsilon)-f(1)}{\varepsilon}.$$

3. (a) Rédiger une fonction `S(n,x)` calculant la somme :

$$S_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}.$$

On pourra se dispenser de calculer la factorielle et la puissance séparément en exprimant le terme d'indice $(i+1)$ dans la somme en fonction du terme précédent d'indice i.

(b) Rédiger une fonction `list_errors(x,N)` renvoyant la liste des quantités :

$$\frac{S_n(x) - \exp(x)}{\exp(x)},$$

où n décrit $[\![1, N]\!]$.

(c) Produire un affichage avec $N = 60$ et différentes valeurs de x. Que constate-t-on ?

Exercice 4 : problème de Dirichlet

Le problème de Dirichlet consiste à trouver une fonction $G : \Omega \longrightarrow \mathbb{R}$ telle que :

- la fonction G soit de classe C^2 sur l'intérieur de Ω et le laplacien ΔG est nul sur l'intérieur de Ω ; on dit que la fonction G est harmonique ;
- la restriction $G_{|\partial\Omega}$ soit égale à la fonction f.

On sait que sous les hypothèses, ce problème admet une seule solution G.

Le but de cet exercice est de procéder à une modélisation discrète de ce problème.

Avant toute chose, il s'agit de discrétiser les notions d'analyse rencontrées. L'ensemble Ω sera inclus dans un carré modélisé par une matrice carrée. Cette matrice carrée notée M est associée au carré discret $I = [\![0, d\ell[\![^2$.
Chaque point intérieur au carré admet quatre voisins plus proches, pour la norme $\|\cdot\|_1$.
On peut ainsi définir l'intérieur d'une partie $J \subset I$. Cet intérieur sera noté $Int(J)$.
Soit K une partie de I.
Une fonction $H : K \longrightarrow \mathbb{R}$ admet pour laplacien discret la fonction définie sur $Int(K)$:

$$\Delta H : \left| \begin{array}{ccc} Int(K) & \longrightarrow & \mathbb{R} \\ (i,j) & \longmapsto & \dfrac{H(i+1,j) + H(i-1,j) + H(i,j+1) + H(i,j-1) - 4H(i,j)}{4} \end{array} \right. .$$

Autrement dit, une fonction discrète harmonique $H : K \longrightarrow I$ est une fonction telle que pour tout point p appartenant à $Int(K)$, la quantité $H(p)$ est toujours égale à la moyenne des valeurs de la fonction H prise sur les quatre voisins du point p.

Implémentation

1. Importer le module pylab. Définir une fonction Contour(X,Y,N) qui prend en argument les fonctions $t \longmapsto X(t)$ et $t \longmapsto Y(t)$ définissant l'arc paramétré Γ, ainsi qu'un entier $N \geqslant 2$ et qui renvoie une matrice M dont les caractéristiques sont les suivantes :
   - on note $LX = X([0, 2\pi])$ et $LY = Y([0, 2\pi])$, en utilisant la syntaxe linspace(0,2*pi,10000) par exemple ; ainsi on utilisera les syntaxes :
     ```
     LX=[X(t) for t in linspace(0,2*pi,10000)]
     LY=[Y(t) for t in linspace(0,2*pi,10000)]
     ```
   - on note $x_{min} = \min(LX)$, $x_{max} = \max(LX)$ et de même pour LY
   - on pose $marge_x = N/10$ et $marge_y = N/10$, puis $dx = x_{max} - x_{min}$ et $dy = y_{max} - y_{min}$ sont les diamètres des ensembles LX et LY
   - on définit deux nouvelles listes $newLX$ et $newLY$ où par exemple $newLX[t] = \left\lfloor marge_x + \dfrac{N}{dx} \times (LX[t] - x_{min}) \right\rfloor$ et de même pour $newLY$
   - la matrice M est initialisée par une matrice de format $\lfloor 1.2 * N \rfloor \times \lfloor 1.2 * N \rfloor$ et dont tous les coefficients valent 1
   - on remplace tous les éléments de M de positions $(i, j) \in newLX \times newLY$ par 0.
2. Définir une fonction Visualisation(M,titre="") qui permet de visualiser la matrice M grâce aux syntaxes
   ```
   figure()
   axis('off')
   title(titre)
   imshow(M,cmap='jet',interpolation='nearest')
   colorbar()
   show()
   ```
3. Définir une fonction Voisins(M,i,j) qui renvoie la liste de toutes les positions voisines dans la matrice M de la case $[i, j]$. On rappelle que si la case $[i, j]$ est intérieure, alors cette case dispose de quatre cases voisines. Afin de ne pas multiplier inutilement les cas, le plus simple est d'élaborer quatre cas qui gèreront les effets de bords, à savoir les cas i>0, j>0, i<dx-1 et j<dy-1 si on a utilisé la syntaxe dx,dy=M.shape.
4. Définir une fonction Composante(M,point,nbre) qui à partir d'une matrice M composée de 0 et de 1 uniquement et à partir d'un point *point* calcule la composante connexe de *point* dans M parmi uniquement toutes les cases $[i, j]$ pour lesquelles $M[i, j] = 1$. La fonction renvoie une matrice de même format que la matrice M de départ où seules les cases de la composante de *point* ont été modifiées et remplacées par la quantité *nbre*. Pour calculer les cases qui figurent dans cette composante connexe, le plus simple est d'incorporer les cases voisines de $[i, j]$ puis de les marquer, puis marquer ces cases nouvellement marquées, ainsi de suite, jusqu'à ce qu'il n'y ait plus de nouvelles cases marquées. Si le point *point* est associée à une case où figure 0 dans la matrice M, on renvoie la liste vide.
5. Faire des essais des fonctions précédentes avec :

- l'arc $\Gamma : t \longmapsto \Big(\cos(t) \times (3 + 2\cos(2t)), \sin(t) \times (3 + 2\cos(3t + 2)\Big)$
- $N = 100$ et $nbre = 2$
- visualiser la matrice faisant apparaître la composante connexe du point $[50, 50]$ intérieur à l'arc.

6. Définir une fonction Spheres_Int(X,Y,N,point) qui à partir d'un arc paramétré associé à X et Y, d'un entier N et d'une case point, calcule la composante de ce point dans la matrice de contour M, puis calcule la liste $Spheres$ dont les occurrences sont encore des listes :
   - on fait calculer la liste Res des cases intérieures à l'arc grâce à une fonction précédente
   - le premier élément de la liste $Spheres$ est l'ensemble des cases $[i, j]$ appartenant à Res dont au moins un voisin est sur le contour
   - le deuxième élément de la liste $Spheres$ est l'ensemble des cases $[i, j]$ appartenant à $Res \setminus Sphere[0]$ dont au moins un élément voisin est dans $Sphere[0]$
   - etc. jusqu'à épuiser tous les points de l'intérieur du domaine délimité par le contour.

   On renverra alors une matrice $Matbis$ de même format que M et telle que les coefficients des cases sur le contour à l'extérieur du contour restent inchangées (valant 0 ou 1 dans ce cas précis) et une case $[i, j]$ est dans $Spheres[r]$ sera associée à $Matbis[i, j] = -r - 1$. On fera une visualisation de ce que cela donne avec l'arc déjà mis en place et $N = 100$.
7. Rédiger une fonction Moyenne(L) qui calcule la moyenne des occurrences numériques de la liste non vide L.
8. Rédiger une fonction Successeur(M) qui à partir d'un tableau M dont les occurrences sont numériques et associées à un certain contour et donc à une liste de sphères Spheres mises en mémoire à travers la syntaxe Spheres_Int(M)[1] renvoie une matrice $Mtemp$ de même format que la matrice M et construite comme suit :
   - si une case $[i, j]$ est à l'extérieur du contour – hors des sphères intérieures, alors $Mtemp[i, j] = M[i, j] = 1$
   - si une case $[i, j]$ est sur le contour – dans Spheres[0], alors $Mtemp[i, j] = F(i, j)$
   - pour les cases $[i, j]$ intérieures au contour – dans Spheres[1 :], on procède comme suit :
     pour chaque k entre 1 et $len(Spheres)$, pour chaque case $[i, j]$ de $Spheres[k]$, on pose :

     ```
     Mtemp[i,j]=Moyenne([M[x[0],x[1]] for x in Spheres[k]])
     ```

9. Faire une visualisation avec X et Y décrits plus haut, $N = 100$, la fonction :

$$F : (i, j) \longmapsto 10 \ \sin\left(\frac{i + j}{10}\right)$$

en implémentant l'animation via le script suivant :

```
def Animation(F,X,Y,N) :
    global Spheres

    M=Contour(X,Y,N)
    Spheres=Spheres_Int(X,Y,N,point)[1]
    contour=Spheres[0]
    L_val=[]
    for pos in contour :
        i,j=pos
        M[i,j]=F(i,j)

    compteur=0

    # mise en place de l'animation
    ion()
    figure(1)
    axis('off')
    image=imshow(M,cmap="jet",interpolation="nearest")
    colorbar()
    show()
    pause(0.05)
    while fignum_exists(1)  :
        M=Successeur(M)
        compteur+=1
        image.set_data(M)
        title("Après "+str(compteur)+" étapes.")
        axis('off')
        image.changed()
        draw()
        pause(0.05)
        show()
    ioff()
```

10. Expliquer en quoi l'algorithme décrit plus haut permet une résolution approchée du problème de Dirichlet.
11. Faire un autre essai où le contour est un carré de côté 1×1 décrit dans le sens trigonométrique et lorsque $F : (i, j) \longmapsto i - j$.

Exercice 5 : résolution approchée par la méthode d'Euler

On considère un pendule de longueur ℓ, où une masse ponctuelle m évolue dans le temps, sans frottement de l'air, dans le champ de pesanteur terrestre supposé uniforme $\vec{g}$.

En notant θ l'angle formé par le pendule par rapport à la verticale, il est facile d'obtenir l'équation du mouvement :

$$\ddot{\theta} + \frac{g}{\ell} \sin\theta = 0.$$

Les conditions initiales sont par exemple $\theta_0 = 1\ rad$ et la vitesse angulaire initiale $\dot{\theta}_0 = 0\ rad \cdot s^{-1}$.
On présente ici la ***méthode de résolution approchée d'Euler explicite***, la résolution se faisant par rapport aux conditions initiales présentées ci-dessus. On posera pour simplifier les notations la pulsation propre :

$$\omega_0 = \sqrt{\frac{g}{\ell}},$$

de sorte que l'équation différentielle devient :

$$\ddot{\theta} + \omega_0^2 \sin\theta = 0.$$

Voici en quoi consiste la méthode de résolution approchée :

- $\longrightarrow$ poser la liste $LT = [t_0 = 0]$
- $\longrightarrow$ poser les deux listes $LX = [\theta_0]$ et $LV = [\dot{\theta}_0]$ correspondant aux conditions initiales
- $\longrightarrow$ se donner un réel t_1 strictement positif : on va résoudre approximativement sur le domaine temporel $[t_0 = 0, t_1]$
- $\longrightarrow$ se donner un pas $h > 0$ de discrétisation
- $\longrightarrow$ tant que le dernier élément de la liste LT est inférieur strictement à t_1, faire le bloc d'instructions :
  - $\triangleright$ poser $t = LT[-1]$ le dernier terme de la liste LT des temps, poser $\theta = LX[-1]$ le dernier terme de la liste LX des angles et $\omega = LV[-1]$ le dernier terme de la liste LV des vitesses angulaires
  - $\triangleright$ ajouter $t + h$ à la liste LT
  - $\triangleright$ ajouter $\theta + h \cdot \omega$ à la liste LX
  - $\triangleright$ ajouter $\omega - h \cdot \omega_0^2 \cdot \sin\theta$ à la liste LV.
- $\longrightarrow$ en sortie de boucle, la liste LT est une subdivision approchée du segment $[t_0, t_1]$ avec un pas de h, la liste LX donne les valeurs approchées des angles et la liste LV donne les valeurs approchées des vitesses angulaires.

1. Rédiger une fonction Euler(t0,t1,h,omega0,theta0,dottheta0) renvoyant la liste

$$[LT, LX, LV],$$

où les listes LT, LX et LV sont présentées dans l'algorithme.

2. Tracer le portrait de phase $\dot{\theta}$ en fonction de θ pour les paramètres suivants :

$$t_0 = 0,\ t_1 = 30,\ \omega_0 = 1,\ \theta_0 = 1,\ \dot{\theta}_0 = 0.$$

On fera apparaître en un point d'une couleur différente la condition initiale dans ce portrait de phase.

Essayer avec différents pas de discrétisation $h = 0.1$, puis $h = 0.01$, puis $h = 0.001$ et enfin $h = 10^{-4}$ par exemple.

3. Tracer la trajectoire θ et $\dot{\theta}$ en fonction du temps pour ces mêmes paramètres. On mettra une légende.
4. Interpréter les résultats.

Exercice 6 : algorithme k-means clustering

Principe de partitionnement

Distance dans $\mathbb{R}^d$

Considérons un entier $d \in \mathbb{N}^*$. On rappelle que si a et b sont deux points de $\mathbb{R}^d$, en posant

$$a = (a_1, \cdots, a_d) \text{ et } b = (b_1, \cdots, b_d),$$

alors la distance euclidienne entre ces deux points vaut :

$$\delta(a, b) = \|a - b\| = \sqrt{\sum_{i=1}^{d} (a_i - b_i)^2}.$$

Germe et diagramme de Voronoï

On considère dans $\mathbb{R}^d$ un ensemble fini de points :

$$S = \Big\{\chi_1, \cdots, \chi_s\Big\}.$$

Soit ρ un point de $\mathbb{R}^d$.
On appelle ***germe du point ρ dans l'ensemble S***, un point $\chi_i \in S$ le plus proche du point ρ :

$$\delta(\rho, \chi_i) = \min\Big\{\delta(\rho, \chi_j) \ ; \ j \in [\![1, s]\!]\Big\}.$$

On appelle ***diagramme de Voronoï associé à l'ensemble S***, le partitionnement de l'espace $\mathbb{R}^d$ en régions $\mathscr{R}_i$ telles que :

$$\forall i \in [\![1, s]\!], \ \mathscr{R}_i = \Big\{\rho \in \mathbb{R}^d \mid \text{ un germe du point } \rho \text{ est } \chi\Big\}.$$

Ce partitionnement comporte des domaines convexes intersections de demi-espaces bordés par des hyperplans affines médiateurs de segments $[\chi_k, \chi_\ell]$.

Centroïde d'un paquet de points

Soit $P = \Big\{\chi_1, \cdots, \chi_m\}$ un ensemble fini de points.
On appelle ***centroïde du paquet*** P, l'isobarycentre de l'ensemble P. Autrement dit, il s'agit du point :

$$g = \frac{1}{m}\sum_{i=1}^{m} \chi_i.$$

Méthode k-means : principe

Étant donné un ensemble fini de points $S = \Big\{\chi_1, \cdots, \chi_s\Big\}$ de l'espace $\mathbb{R}^d$ puis un entier $k \geqslant 1$ et $k \leqslant s$, l'objectif est de partitionner l'espace en k régions qui contiendront des points de S proches les uns des autres.
Le principe est le suivant :

- $\longrightarrow$ choisir une valeur de *seuil* strictement positive, en pratique, une valeur petite
- $\longrightarrow$ choisir k points différents dans l'ensemble S ; on note $\rho_1, \cdots, \rho_k$ ces points, puis $C = \Big\{\rho_1, \cdots, \rho_k\Big\}$
- $\longrightarrow$ pour chaque entier $i \in [\![1, k]\!]$, on forme le paquet :

$$P_i = \Big\{x \in S \mid \text{ un germe de } x \text{ dans } C \text{ est égal à } \rho_i\Big\}$$

  Chaque paquet P_i est non vide car contient ρ_i.
- $\longrightarrow$ former les centroïdes g_i de chaque paquet P_i
- $\longrightarrow$ poser le nouvel ensemble de points :

$$C' = \Big\{g_1, \cdots, g_k\Big\}$$

- $\longrightarrow$ calculer l'écart quadratique $\varepsilon(C, C')$ entre les ensembles C et C' en calculant :

$$\varepsilon(C, C') = \sum_{i=1}^{k} \delta^2(g_i - \chi_i).$$

- $\longrightarrow$ effectuer l'affectation $C \longleftarrow C'$
- $\longrightarrow$ recommencer les calculs à partir de la première étape avec ce nouvel ensemble C, jusqu'à à obtenir :

$$\varepsilon(C, C') < \textit{seuil}.$$

Méthode k-means : explications

La suite d'ensembles C est une suite convergente, les suites de points (ρ_1), $\cdots$,(ρ_k) prenant leurs valeurs dans un compact de $\mathbb{R}^d$ et en fait par le calcul des centroïdes, ces suites sont convergentes.

Lorsque la condition $\varepsilon(C, C') < \textit{seuil}$ est réalisée, on estime que les ensembles C et C' sont pratiquement égaux. Autrement dit, les paquets P_i se stabilisent jusqu'à être inchangés. On obtient dans ce cas des centroïdes g_i délimitant des zones P_i associés à des points proches les uns des autres, en tout cas, proches de g_i.
Les paquets stabilisés correspondent à des paquets de points proches les uns des autres. Ces paquets sont appelés ***clusters de points***.

Applications sur la segmentation d'image en k clusters de points

On considère une image en couleurs ou en noir et blanc, associée à un tableau M.
On note dx et dy le nombre de lignes et de colonnes de ce tableau.
Pour tout $(i, j) \in [\![0, dx-1]\!] \times [\![0, dy-1]\!]$, le vecteur $M[i, j]$ est soit un vecteur de $\mathbb{R}^1$ (si l'image est en noir et blanc), soit un vecteur dans $\mathbb{R}^3$ (si l'image est en couleurs).
On peut donc considérer que l'algorithme précédent appliqué aux points :

$$S = \Big\{ M[i, j] \ ; \ (i, j) \in [\![0, dx-1]\!] \times [\![0, dy-1]\!] \Big\}$$

permet de calculer des clusters de points proches les uns des autres. Autrement dit, on partitionne l'image en k paquets de pixels de couleurs proches.
On transforme l'image en étiquetant à chaque cluster une couleur différente et cela permet de visualiser des zones de l'image à pixels plus ou moins identiques.

Questions

1. Rédiger une fonction `V(a,b)` qui à partir de deux tableaux-lignes de même format a et b calcule la quantité $\delta^2(a, b)$.
2. Rédiger une fonction `Moyenne(M,S)` qui à partir d'un tableau M et d'un ensemble S de case $[i, j]$ renvoie le centroïde du paquet $P = \Big\{ M[i, j] \ ; \ [i, j] \in S \Big\}$. On pourra utiliser la syntaxe `m=zeros_like(M[0,0])` pour initialiser un tableau rempli de zéros et de même format que $M[0, 0]$.
3. Rédiger une fonction `IsIn(t,L)` testant si un vecteur t appartient à une liste L de tableaux. On utilisera la syntaxe `all(t,u)` pour tester si les deux tableaux t et u sont égaux.
4. Rédiger une fonction `Choisir(M,k)` qui à partir d'un tableau M et d'un entier k strictement positif choisit aléatoirement k vecteurs $M[i, j]$ différents dans le tableau M.
5. Rédiger une fonction `Paquets(M,L_k)` qui à partir d'un tableau M et d'un paquets L_k de vecteurs de même format que $M[0, 0]$:
   $\longrightarrow$ parcourt la liste $[i, j]$ des cases du tableau M
   $\longrightarrow$ pour chacune de ces cases $[i, j]$, on calcule un entier m tel que le point $L_k[m]$ est le plus proche de $M[i, j]$ parmi tous les vecteurs de L_k

$\longrightarrow$ renvoie une liste Res initialement vide dont les occurrences $Res[\ell]$ correspondent à la liste des cases $[i,j]$ tel que le point le plus proche de $M[i,j]$ parmi les points de L_k est le point $L_k[\ell]$

6. Rédiger une fonction Successeur(M,L_Paquets) qui à partir d'un tableau M et d'une liste de paquets de cases $[i,j]$ renvoie la liste des centroïdes des paquets L_Paquets$[\ell]$, lorsque ℓ décrit tous les index de L_Paquets
7. Rédiger une fonction Algo_K_Means(M,k,seuil) qui renvoie une liste de k clusters associés à l'algorithme présenté ci-dessus.
8. On considère la liste des couleurs

   L_couleurs=[(0.8,0.9,0.8),(0.9,0.8,0.3),(0.4,0.7,0.8),(0.9,0.4,0.2)].

   En utilisant le module os, appliquer l'algorithme k-means précédent avec $k=4$ pour segmenter l'image « panda.png » ou l'image « toucans.png », avec $k=3$ ou $k=4$.

   On mettra en place une procédure de visualisation de l'image initiale et de l'image segmentée avec un titre indiquant le niveau k de segmentation.