Master 1 Septembre 2019

Initiation au calcul numérique avec Python – TP

---

1 Représentation des nombres réels en machine et erreurs d'arrondis

1.1 Représentation des nombres réels en machine

Dans un ordinateur, les nombres réels sont stockés en utilisant la méthode 'virgule flottante' : tout nombre réel est représenté dans une base b ($b = 2$ pour un ordinateur), par son signe ($+$ ou $-$), par une mantisse m et par un exposant e :

$$x = \pm m\, b^e \tag{1}$$
$$m = d.dd\cdots d \tag{2}$$
$$e = dd\cdots d \tag{3}$$

où $d \in \{0, 1, \cdots b-1\}$ représente un chiffre. Pour rendre la représentation unique, on utilise une mantisse normalisée : le premier chiffre avant le point décimal de la mantisse est non nul.

Exemples.

- *Le nombre $x = 0.625$ sera stocké en binaire $x = 1.01 \cdot 2^{-1}$ ($m = 1.01$, $e = -1$, $b = 2$). En effet, $y = \frac{1}{2} + \frac{1}{8}$.*
- *Le nombre $y = -0.000345678$ s'écrira en utilisant $b = 10$ comme $y = -m\,10^e$ où $m = 3.45678$ et $e = -4$. Comme la mantisse peut stocker au maximum n chiffres, si $n \geq 6$, le nombre est stocké de manière exacte. Au contraire si $n < 6$, alors le nombre y n'est pas stocké de manière exacte. Par exemple si $n = 4$, alors $m = 3.456$. Il y a une **erreur d'arrondi**.*
- *Le nombre $z = \frac{1}{3}$ s'écrit, en décimal, $0.3333333\cdots$. Comme son développement est infini, il est **impossible** de stocker z en base 10 sans faire d'erreur d'arrondi. Le même problème se pose en base 2, où z s'écrit $0.0101010101\cdots$ (en effet, $\frac{1}{3} = \frac{\frac{1}{4}}{1-\frac{1}{4}} = \frac{1}{4} + \frac{1}{4^2} + \frac{1}{4^3} + \cdots$). Attention, le nombre $\frac{1}{5}$ s'écrit en décimal 0.2, mais en binaire son développement est infini : seuls les nombres dyadiques (de la forme $\frac{k}{2^n}$) ont un développement fini en base 2.*

La mantisse m est toujours stockée avec une **précision limitée** (correspondant à un certain nombre de chiffres significatifs). En effet, la mémoire d'un ordinateur n'est pas infinie. Par conséquent, **les nombres ne sont pas toujours stockés de manière exacte**. En particulier, sur un ordinateur (utilisant la base 2) les nombres irrationnels (π, $\sqrt{2}$, ...), mais aussi tous les nombres non dyadiques ($\frac{1}{3}$, $\frac{1}{5}$, $\frac{2}{3}$,...), ne peuvent pas être stockés de manière exacte. De même l'exposant e ne peut excéder une certaine valeur, si bien que les nombres trop grands, ou trop proches de zéro, ne peuvent être représentés.
En Python, le stockage des nombres suit le format IEEE double précision (nombres réels stockés en binaire sur 64 bits).

1.2 Erreurs d'arrondis : perte de précision numérique

Que vaut, pour Python, `0.2+0.1-0.1-0.2` ?

Exercice 1 : Précision machine

Quand on utilise dans Python des nombres réels (type `float`), on travaille avec une précision limitée. Par conséquent, il existe un entier k tel que

- $1 + 2^{-k}$ " $\neq$ " 1
- pour tout $q > k$, $1 + 2^{-q}$ " $=$ "1.

L'égalité précédente doit se comprendre au sens suivant : le nombre $1 + 2^{-q}$ et le nombre 1 ont même représentation en Python. Autrement dit, Python ne fait pas la différence entre 1 et $1 + 2^{-q}$ dès lors que $q > k$. On appelle **précision machine** le nombre $\varepsilon = 2^{-k}$.

L'objectif de cet exercice est de calculer la **précision machine**.

1. Concevoir et coder en Python un algorithme pour calculer la **précision machine**. On pourra utiliser une méthode de dichotomie : on part de $\eta = 2^0 = 1$, et tant que $\eta + 1$ est différent de 1, on divise η par 2.

Exercice 2 : Erreurs de compensation

On souhaite observer numériquement la convergence suivante :

$$\lim_{n\to+\infty}\left(1+\frac{1}{n}\right)^n = e.$$

1. Calculer la quantité $\left|(1+\frac{1}{n})^n - e\right|$ pour $n = 10^k$ et k allant de 1 à 16.
2. Expliquer le phénomène observé.

Exercice 3 : Instabilités numériques

Soit $S_n = \displaystyle\int_0^\pi \left(\frac{x}{\pi}\right)^{2n} \sin x\, dx$. Puisque l'intégrant est une fonction positive sur l'intervalle considéré, la suite $(S_n)_{n\in\mathbb{N}}$ est positive.

1. Justifier que la suite $(S_n)_{n\in\mathbb{N}}$ tend vers 0.
2. Vérifier que $S_0 = 2$ et montrer que l'on a la relation de récurrence

$$S_n = 1 - \frac{2n(2n-1)}{\pi^2} S_{n-1} \text{ pour } n \in \mathbb{N}^*. \tag{4}$$

3. Écrire une fonction Python `Calcul_Sn` qui renvoie la suite $(S_k)_{0\leq k\leq n}$ (sous forme de liste ou de tableau), et calculer $S_0, \ldots, S_{16}$. Qu'observe-t-on ?

2 Représentation graphique avec matplotlib

Le module `matplotlib.pyplot` permet de représenter des graphes de fonctions. Dans les programmes suivants, on commencera par charger ce module ; par exemple, texter le code suivant :

```
from math import *
import numpy as np
import matplotlib.pyplot as plt

vx=np.linspace(0,2*pi,1000)
vy1=[sin(x) for x in vx]    # ou : vy1=np.sin(vx)
vy2=[cos(x) for x in vx]

plt.figure(1)    # ouvre la fenêtre 1
plt.clf()   # efface le contenu éventuel
plt.plot(vx,vy1,'r',label='cos')    # affiche un graphe
plt.plot(vx,vy2,'--b',label='sin')    # puis un autre
plt.title('Graphes de sinus et cosinus')
plt.legend() # affiche la légende (label donnés ci-dessus)
plt.show() # met à jour l'affichage
```

Quelques explications : la commande `plot` de `matplotlib` (très inspirée de la commande `plot` de Matlab) permet de tracer des courbes : si `vx` et `vy` sont des listes ou des vecteurs (`numpy.array`) de taille n, la commande `plot(vx,vy)` va tracer les points de coordonnées $A_i = (\texttt{vx[i]}, \texttt{vy[i]})$ ainsi que les segments $[A_iA_{i+1}]$. De nombreuses options peuvent être associées à cette commande (type de trait, type de points, couleurs des traits et des points ...). La syntaxe la plus courte est `plot(a,b,'-r')`, où le tiret `'-'` correspond au type de trait (ici, continu) et le `'r'` à la couleur (ici, rouge). On pourra tester les types de trait `-`, `.`, `+`, `:`,`-.` par exemple, et les couleurs `rgbkymc`. Si elle n'est pas précisée, la couleur change automatiquement (dans une courte liste) à chaque tracé.
Pour obtenir de l'aide sur cette commande, on pourra taper `help(plt.plot)` dans la console ou se référer à la documentation de matplotlib en ligne.

Remarquons que pour représenter le graphe d'une fonction $f : [a,b] \to \mathbb{R}$, on sera ainsi amené à **choisir une subdivision** $x_1, \ldots, x_n$ de $[a,b]$ (souvent, on prendra une subdivision à pas constant, avec `x=np.linspace(a,b,n)`), puis on calculera le vecteur $y_1 = f(x_1), \ldots, y_n = f(x_n)$ et on représentera `plot(x,y)`, qui interpole linéairement entre les points (x_i, y_i). Ainsi, **la représentation graphique d'une fonction a une précision limitée**, à la fois par les capacités graphiques matérielles (résolution de l'écran), mais aussi par le nombre de points calculés.

Exercice 4 : Graphisme élémentaire 2D

Les codes associés à cet exercice pourront être sauvegardés dans un script `Exercice1.py`.

1. Tracer le cercle de centre C de coordonnées $(x_C, y_C) = (2, 4)$ et de rayon $r = 1$ en **rouge** et en **pointillés** en utilisant une équation paramétrique du cercle :

$$\begin{cases} x(\theta) = x_C + r\cos(\theta) \\ y(\theta) = y_C + r\sin(\theta) \end{cases} \qquad \theta \in [0, 2\pi].$$

Pour vous assurer que les axes sont à la même échelle, utiliser la commande `plt.axes(aspect='equal')` avant la commande `plot`.

2. Ajouter le titre *'Cercle de rayon* 1 *et de centre* $(2, 4)$*'*.

3. Tracer sur le même graphique un second cercle de même centre et de rayon 0.5 en noir avec un trait épais d'épaisseur 3 (pour varier l'épaisseur, ajouter l'option `linewidth=3` dans la commande `plot`)

4. Sur une troisième figure, tracer le graphe de la fonction $f : x \mapsto f(x) = 3x^2 + \dfrac{\ln\left((x-\pi)^2\right)}{\pi^4} + 1$ dans l'intervalle $[\pi - 1, \pi + 1]$. Une remarque sur le comportement en π ?

5. Tracer dans une même fenêtre mais sur deux graphes différents les fonctions $\cos x$ et $\sin x$ entre 0 et 2π. On utilisera pour cela la commande `subplot(m,n,k)` qui découpe une fenêtre en `m` lignes `n` colonnes de sous-fenêtres et sélectionne la `k`-ième. Ici, on utilisera donc respectivement `subplot(2,1,1)` et `subplot(2,1,2)` avant le premier et le deuxième graphe.

3 Nombres pseudo-aléatoires

De nombreux domaines d'application utilisent des nombres aléatoires : sécurité informatique (génération automatique d'identifiants, de clés secrètes), méthodes d'optimisation dans des espaces de grande dimension (recuit simulé, algorithmes génétiques), simulations numériques de systèmes complexes (physique, ingénierie, finance, assurance, météo...), jeux vidéos (paysages aléatoires, intelligence artificielle,...),...
A priori, vouloir utiliser un ordinateur pour obtenir des nombres aléatoires apparaît paradoxal, sinon impossible : par définition, un nombre aléatoire n'est pas prévisible, tandis que l'ordinateur ne peut appliquer qu'une formule ou un algorithme prédéfinis. Il existe cependant des algorithmes fournissant des suites de nombres ayant **de façon approximative** certaines propriétés d'une suite de variables aléatoires indépendantes et de même loi. On parle de nombres **pseudo-aléatoires**.
On s'attend à ce qu'une suite pseudo aléatoire suive la loi des grands nombres, et soit « bien mélangeante ». Une définition plus précise est délicate, et dépendrait en fait de l'application voulue : parfois, on souhaite simplement une suite de valeurs « bien répartie » dans un ensemble, sans s'inquiéter de la dépendance entre termes successifs, tandis que d'autres contextes exigent des contraintes d'indépendance plus importantes (par exemple en cryptographie). De plus, si on a besoin d'une grande quantité de nombres aléatoires, alors il faut un meilleur générateur, qui peut être plus coûteux en temps de calcul.
Dans le module `numpy.random` de Python, le générateur de nombres aléatoires prend la forme de la "fonction" `random()` (sans paramètre), qui est particulière car son résultat est un nombre réel dans $[0, 1[$ qui change a priori à chaque fois qu'on appelle cette fonction. On fera l'hypothèse (informelle) suivante :

> Les différents appels à `random` renvoient une réalisation "générique" d'une suite de variables aléatoires indépendantes et de loi uniforme dans $[0, 1]$.

Autrement dit, il existe une suite $(U_n)_{n\geq 1}$ de variables aléatoires (c'est-à-dire de fonctions mesurables $U_n : \Omega \to \mathbb{R}$ définies sur un espace de probabilités $(\Omega, \mathcal{F}, P)$) indépendantes et suivant chacune la loi uniforme sur $[0, 1]$, et il existe un $\omega \in \Omega$ tels que `random` renvoie $U_1(\omega)$, puis $U_2(\omega)$, etc.
La valeur ω est la **graine** du générateur : si on connaît ω, la suite $(U_n(\omega))_{n\geq 1}$ est entièrement connue ! Cela est utile pour reproduire une simulation avec les mêmes "nombres aléatoires". Dans l'hypothèse ci-dessus, on a précisé "générique", pour signifier que ω réalise ("en pratique") n'importe quel événement presque sûr. On pourra donc utiliser `random` pour observer des propriétés vraies presque sûrement, notamment la loi forte des grands nombres.
Dans le module `numpy.random`, la commande pour fixer la graine égale à n (entier positif) est `seed(n)`. On peut aussi vouloir laisser Python choisir la graine de façon plus "aléatoire" (en fait la graine est choisie en utilisant l'heure, en millisecondes) avec `seed()`, auquel cas le résultat du premier appel à `random` est à peu près imprévisible.

Exercice 5 : Générateur et graine, utilisation de `random` et `seed`

Commencer par entrer dans la console

```
from numpy.random import random,seed
```

1. Dans la console, exécuter `seed(1)`, puis exécuter `random()` plusieurs fois et observer les résultats. Exécuter à nouveau `seed(1)` et faire à nouveau appel à `random()` : que remarque-t-on ? Recommencer en remplaçant `seed(1)` par `seed(2)`.
2. Recommencer en remplaçant `seed(1)` par `seed()`, et réessayer plusieurs fois : que remarque-t-on ? Est-ce que utiliser ensemble `seed()` et `random()` à chaque fois que l'on veut un nombre aléatoire (par exemple dans une boucle) est une bonne méthode pour obtenir une suite de nombres "vraiment" aléatoires ? *(Non)*

Exercice 6 : Un exemple de loi discrète

À partir de nombres qui suivent la loi uniforme dans $[0,1]$, on verra que l'on peut obtenir des variables aléatoires réelles, ou dans $\mathbb{R}^d$, suivant toute loi donnée. On dira que l'on **simule** une loi de probabilité lorsque l'on fait produire à Python un nombre aléatoire suivant cette loi. Voyons pour le moment un exemple de loi discrète.
1. Écrire une fonction `signe_aleatoire(p)` qui renvoie un nombre aléatoire qui vaut 1 avec probabilité `p`, et -1 avec probabilité $1-$`p`. *Indication : si U suit la loi uniforme sur $[0,1]$, quelle est la probabilité que U soit inférieur à p ?*
2. Écrire une fonction `signes_aleatoires(n,p)` qui renvoie un vecteur (de type `np.array`) de n nombres aléatoires donnés par la fonction précédente.
3. Qu'illustre-t-on par le code suivant ? :

```
N=1000
vn=np.arange(1,N+1)
plt.plot(vn,np.cumsum(signes_aleatoires(N,0.5))/vn)
plt.show()
```

Exercice 7 : Générateurs de nombres aléatoires par congruence linéaire

Pour bien comprendre comment `random` fonctionne, écrivons notre propre générateur de nombres pseudo-aléatoires, bien plus simple que celui de Numpy. Les méthodes du type suivant ont été utilisées jusque dans les années 90. On définit la suite $(X_n)_{n\geq 0}$ par récurrence, par la donnée de $X_0 = \omega \in \{1,\dots,2^{31}-1\}$ (c'est la graine) puis

$$\text{pour tout } n \geq 0, \qquad X_{n+1} = 16807\ X_n \pmod{2^{31}-1}.$$

Alors les valeurs $(X_n)_{n\geq 1}$ se comportent approximativement comme des entiers aléatoires uniformément choisis entre 0 et $2^{31}-1 = 2147483647$, et donc $U_n = 2^{-31}X_n$ suit approximativement la loi uniforme sur $[0,1]$.
On peut montrer que 16807 est un générateur du groupe cyclique $(\mathbb{Z}/(2^{31}-1)\mathbb{Z})^*$ (le nombre $2^{31}-1$ est premier), ainsi la suite $X_0, X_1, \dots, X_{N-1}$ parcourt une fois chacune des valeurs de $1, 2, \dots, N-1$, où $N = 2^{31}-1$, et $X_N = X_0$.
1. On souhaite écrire une fonction `u=alea()` qui renvoie un nombre aléatoire uniforme dans $[0,1]$ à l'aide de cette méthode, et une fonction `graine(n)` qui donne à la graine la valeur `n`. Reproduire les codes suivants et les tester quelques fois (les résultats ont-ils bien l'air aléatoires ?), et comprendre comment il se fait que `alea()` renvoie des valeurs différentes alors qu'il n'y a pas de paramètre.

```
def alea():
    global X
    X=(X*16807)% 2147483647
    return X/2147483648.
```

```
def graine(n):
    global X
    X=n
```

2. Si l'on a besoin de plus que $2^{31}(\simeq 2\cdot 10^9)$ nombres aléatoires, peut-on utiliser ce générateur ?
3. Représenter un histogramme de 1000 valeurs fournies par la fonction `alea`. On utilisera la fonction `plt.hist(x,bin=k)` où `x` est le vecteur des données, et `k` est le nombre de classes de l'histogramme. Que met-on ainsi en évidence ?
4. Écrire une fonction `points2D(n)` qui représente, sur un graphique, n points $(x_1,y_1), (x_2,y_2), \dots, (x_n,y_n)$, où $x_1, y_1, x_2, y_2, \dots$ sont aléatoires, indépendants et uniformes dans $[0,1]$, obtenus grâce à la fonction `alea`. Apprécier visuellement la répartition des points.
5. Écrire une fonction `points3D(n)` qui représente, sur un graphique, n points (x_1,y_1,z_1), $(x_2,y_2,z_2), \dots, (x_n,y_n,z_n)$, où $x_1, y_1, z_1, x_2, y_2, z_2, \dots$ sont aléatoires, indépendants et uniformes dans $[0,1]$, obtenus dans cet ordre grâce à la fonction `alea`. *On utilisera les commandes suivantes :*

```
from mpl_toolkits.mplot3d import Axes3D
Axes3D(plt.figure(3))
```

puis `plt.plot(X,Y,Z,'.')` où `X`, `Y` et `Z` sont des listes ou vecteurs de même taille, comme en 2 dimensions.
6. Entre les années 60 et les années 80, le générateur de nombres aléatoires de certaines bibliothèques FORTRAN très répandues, nommé RANDU, utilisait la récurrence $X_{n+1} = 65539X_n \pmod{2^{31}}$, puis $U_n = 2^{-31}X_n$, en partant d'un $X_0 \in \{1,\dots,2^{31}-1\}$. Il est aujourd'hui considéré comme exemple caractéristique de *mauvais* générateur. Modifier le code précédent, et exécuter `points3D(1000)`. Est-ce que les points semblent bien répartis dans le cube ?