INFORMATIQUE

Représentation des nombres et erreurs d'arrondis

Lorsque l'on met en place une méthode numérique pour résoudre un problème scientifique il importe d'avoir une approche critique des résultats. En ce qui concerne la précision, il est utile de pouvoir apprécier l'erreur produite par la méthode numérique utilisée. Cette erreur est essentiellement de deux types : l'erreur d'arrondi et l'erreur d'approximation.

I Revisite du cours : représentation des nombres réels en machine

Dans une machine, un nombre réel x est codé et stocké dans un format dit en « virgule flottante » : il est représenté dans une base b ($b = 2$ pour un ordinateur), par son signe ($+ ou -$), une mantisse m et un exposant e :

$$x = \pm m\, b^e$$
$$m = d.dd \cdots d$$
$$e = dd \cdots d$$

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

Les exemples qui suivent ne prennent pas en compte l'offset introduit pour le codage signé de l'exposant e suivant la norme $IEE754$[1]
Pour une description plus précise, vous devez vous référer au cours.

- Le nombre $x = 0.625 = \frac{1}{2} + \frac{1}{8}$ est codé en binaire $x = 1.01 \times 2^{-1}$ et $m = 1.01, e = -1, b = 2$.
- En base 10, le principe reste le même. Soit le nombre $y = -0.000345678$ et $b = 10$. En conséquence, $y = -m \times 10^e$ où $m = 3.45678$ et $e = -4$. Considérons une mantisse à n chiffres, deux situations sont à envisagées :

$n \geq 6$, le nombre y est codé de manière exacte.
$n < 6$, le nombre y n'est pas codé de manière exacte. Ainsi, si $n = 4$, alors $m = 3.456$. Il y a une erreur d'arrondi.

- Revenons à un exemple donné en cours avec le codage de $z = \frac{1}{3}$ qui a un développement infini. Il est impossible de coder z en base 10 sans faire d'erreur d'arrondi. Le même problème se pose évidemment en base 2.

Le nombre $\frac{1}{5} = 0.2$ a un développement infini en puissance de 2, son codage en base 2 sera aussi tronqué.
Ces arrondis et ces troncatures des nombres codés ont pour origine la mantisse m qui est toujours codée avec une précision finie. C'est particulièrement vrai pour les nombres irrationnels tels que $\pi, e, \sqrt{2}, \ldots$ et plus généralement pour les nombres non dyadiques (qui ne sont pas de la forme $\frac{k}{2^n}$). Il en est de même pour l'exposant e qui a une dynamique de codage nécessairement limitée, si bien que les nombres trop grands, ou trop proches de zéro, ne peuvent être représentés.
En Python, le codage des nombres est conforme à la norme IEEE 754 en double précision c'est-à-dire sur 64 bits soit 8 octets.

- Le premier bit représente le signe $\pm$ (0 pour un nombre positif, 1 pour négatif)
- Les 11 suivants représentent l'exposant e (entre -1023 et +1024)
- Les 52 derniers codent la mantisse m comprise entre 1 et 2.

Objectif de la séance : plusieurs types et différentes causes caractérisent les erreurs numériques. Nous nous intéresserons d'abord à la précision de la machine, puis plus généralement aux erreurs générées par le codage des nombres en précision finie ainsi que la propagation de ces erreurs (smearing).
En début de votre script, importer la bibliothèque math. En mode préfixé, dir(math) retourne la liste des fonctions de la bibliothèque.
De la même façon, la bibliothèque des fonctions de plot : import matplotlib.pyplot as plt.
En fin de document, vous avez un rappel des fonctions nécessaires pour une représentation correcte de vos résultats.

II Erreurs d'arrondis : perte de précision numérique

Arrondi, troncature, … bref, que vous retourne Python suite au traitement de l'opération 0.2+0.1-0.1-0.2 ?

2-1 : Estimation de la précision machine

Python, mais aussi Matlab, … codent les nombres réels (type float) avec une précision finie. Par conséquent, il existe un entier k tel que :

$a = 1 + 2^{-k}$ est codé sans erreur et pour $q > k$, le codage du réel a est tel que $a = 1 + 2^{-q} = 1$.

L'égalité signifiant que le codage de a sera confondu 1 car 2^{-q} est trop petit pour pouvoir être codé.
On appelle précision machine le nombre $\varepsilon = 2^{-k}$. Sa connaissance est d'une grande importante, ainsi lorsque qu'un résultat X est effectué en N multiplications, on montre que l'erreur relative $\frac{\delta X}{X}$ varie comme $\sqrt{N}\,\varepsilon$.

[1] La norme IEEE754 est un standard pour la représentation des nombres à virgule flottante en binaire.

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

Q1. Proposer un algorithme qui retourne la valeur de la précision machine. Ainsi par exemple, on part de $\eta = 2^0 = 1$, et tant que $\eta + 1$ est différent de 1, on divise η par 2.

En déduire le nombre de décimales exactes.

Quelques règles pour limiter les erreurs d'arrondi :

- Utiliser un système d'unités tel que les valeurs traitées sont de l'ordre de l'unité.
- Éviter de soustraire des nombres comparables, minimiser les soustractions.
- Chercher à sommer par paquet les quantités de même ordre de grandeur.

2-2 Un exemple : une suite qui devrait tendre vers 2, … mais non !

Considérons la suite d'ordre 2 u_n et $n > 2$, un classique pour illustrer notre problème, définie par $u_0, u_1 \in \mathbb{R}$ [Equ. 1] :

$$u_n = 2003 - \frac{6002}{u_{n-1}} + \frac{4000}{u_{n-1}u_{n-2}}$$

On montre par récurrence que le terme général de la suite u_n peut être mis sous la forme suivante [Equ. 2] :

$$u_n = \frac{a + 2^{n+1}b + 2000^{n+1}c}{a + 2^n b + 2000^n c}$$

La coefficients a, b, c permettent de définir les termes u_0, u_1 de la suite u_n. Il est facile de vérifier que si $c \neq 0$, la suite tend vers 2000 et si $c = 0$ et $b \neq 0$ elle tend vers 2.

Ainsi pour $a = b = 1$ et $c = 0 \Longrightarrow u_0 = \frac{3}{2}$ et $u_1 = \frac{5}{3}$ et $\lim_{n\to\infty} u_n = 2$, l'évolution de la convergence de la suite est représentée sur la figure 1.

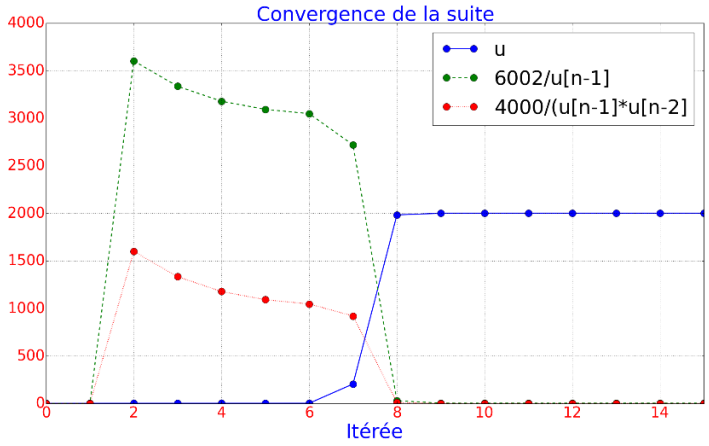


Figure 1 : Evolution de la convergence de la suite u_n

Q2. Proposer le code d'une fonction de nom $cvu(N)$ d'argument N qui retourne les N premiers termes de la suite u_n pour $u_0 = \frac{3}{2}$ et $u_1 = \frac{5}{3}$.

Vous justifierez ce résultat en vous aidant de [Equ. 2].

2-3 Revisite du calcul de la constante π

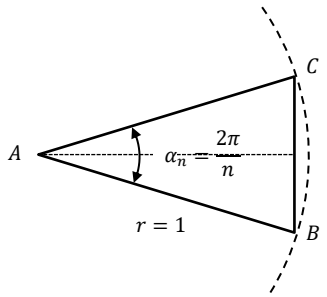
Cet exemple est extrait de [2], [1]. Il est aussi une question du DM1 pour lequel on vous propose de faire le code afin d'étudier la convergence de la suite associée à l'estimation de π. L'énoncé de la question est en partie une reprise du DM1.

Le nombre π est connu depuis l'antiquité, en tant que méthode de calcul du périmètre du cercle ou de l'aire du disque. Le problème de la quadrature du cercle étudié par les anciens Grecs consiste à construire un carré de même aire qu'un cercle donné à l'aide d'une règle et d'un compas. Ce problème resta insoluble jusqu'au 19ème siècle, où la démonstration de la transcendance de π montra que le problème ne peut être résolu en utilisant une règle et un compas.

L'aire d'un cercle de rayon r est aujourd'hui connue, … Il fut un temps où il n'en était rien. Parmi les solutions alors proposées pour l'approcher, une méthode consiste à construire un polygone dont le nombre de côté augmenterait jusqu'à ce qu'il devienne équivalent au cercle circonscrit. C'est Archimède, vers 250 avant J-C, qui appliquera cette propriété au calcul des décimales du nombre π, en utilisant à la fois un polygone inscrit et circonscrit au cercle. Il utilise ainsi un algorithme pour le calcul et parvient à l'approximation de π dans l'intervalle $\left[3 + \frac{10}{71}, 3 + \frac{1}{7}\right]$ en faisant tendre le nombre de côtés jusqu'à 96.

Regardons l'algorithme de calcul par les polygones inscrits. On considère un cercle de rayon $r = 1$ et on note A_n l'aire associée au polygone inscrit à n côtés. En notant $\alpha_n = \frac{2\pi}{n}$, l'aire A_n est égale à n fois l'aire du triangle ABC représentée sur la figure ci-dessous.

Nous avons : $$A_n = n cos\frac{\alpha_n}{2} sin\frac{\alpha_n}{2} = \frac{n}{2}\left(2cos\frac{\alpha_n}{2} sin\frac{\alpha_n}{2}\right) = \frac{n}{2} sin\alpha_n = \frac{n}{2} sin\left(\frac{2\pi}{n}\right)$$



Quadrature du cercle

L'estimation de π repose sur la recherche d'une relation de récurrence sur A_n. Elle est construite simplement à partir de relations trigonométriques élémentaires, soit :

$$sin\frac{\alpha_n}{2} = \sqrt{\frac{1 - cos(\alpha_n)}{2}} = \sqrt{\frac{1 - \sqrt{1 - sin^2(\alpha_n)}}{2}}$$

Choisissons n telle que $n = 2^k$, c'est une relation dyadique qui existe entre deux itérées successifs ; l'angle α_n est divisé par deux alors que le nombre de triangles isocèles est multiplié par deux.

Avec l'aire $A_n = \frac{n}{2} sin\left(\frac{2\pi}{n}\right) \Rightarrow A_{2^k} = 2^{k-1} sin(s_k)$ avec $s_k = sin\frac{2\pi}{2^k}$, il vient la récurrence :

$$sin\frac{2\pi}{2^k} = \sqrt{\frac{1 - \sqrt{1 - sin^2\left(2\frac{2\pi}{2^k}\right)}}{2}} \Leftrightarrow \begin{cases} s_k = \sqrt{\frac{1 - \sqrt{1 - s_{k-1}^2}}{2}} \\ A_{2^k} = 2^{k-1} s_k \end{cases}$$

On a $\lim_{k\to\infty} A_k = \pi$, ce n'est toutefois pas du tout ce que l'on observe sur machine à cause du codage sur un nombre fini de bits.

La récurrence précédente est initialisée par $\alpha_4 = \frac{\pi}{2}$ donc $sin(\alpha_4) = 1$.

Le pseudocode de l'algorithme est le suivant :

Fonction : $\boldsymbol{EstDePi(N)}$

Entrée : N

Sortie : $\mathcal{A}$**, une estimation de la constante** π

Début

$\boldsymbol{s = 1, n = 4}$

Faire tant que

$\boldsymbol{s \leftarrow \sqrt{\frac{1-\sqrt{1-s*s}}{2}}}$

$\boldsymbol{n \leftarrow 2 * n}$

Fin

$\boldsymbol{\mathcal{A} \leftarrow s * n/2}$

Retourner $\boldsymbol{\mathcal{A}}$

Q3. Décliner le code python de la fonction $\boldsymbol{EstDePi(N)}$ d'argument N, le nombre de polygones inscrit dans le disque de rayon 1 au sortir du traitement itératif, qui retourne l'aire $\mathcal{A}_N$ une estimation de la constante π. Interpréter le résultat puis conclure.

Modifier votre algorithme afin qu'il retourne la liste des N premiers termes de la suite et observer son évolution.

III Autour des erreurs dans les traitements numériques

On s'intéresse à la convergence suivante :

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

Q4. Calculer la quantité $\left|\left(1+\frac{1}{n}\right)^n - e\right|$ pour $n = 10^k$ et $k = 1, \dots, 16$. Un représentation semi-logarithmique de l'erreur relative d'estimation de e semble pertinente ...

Q5. Expliquer le phénomène observé.

Soit la suite définie par :

$$S_n = \int_0^{\pi}\left(\frac{x}{\pi}\right)^{2n} sinxdx$$

Remarquons que la suite $(S_n)_{n\in\mathbb{N}}$ est positive puisque la fonction intégrée est positive sur le support.

On démontre la relation de récurrence suivante :

$$S_n = 1 - \frac{2n(2n-1)}{\pi^2}S_{n-1}, n \in \mathbb{N}^*$$

Remarquons que la suite $(S_n)_{n\in\mathbb{N}}$ tend vers 0.

Q6. Vérifier que $S_0 = 2$.

Q7. Ecrire une fonction qui retourne la suite $(S_k)_{0\le k\le n}$ et $n = 30$. Qu'observe-t-on ?

On souhaite analyser l'effet d'accumulation d'erreurs associées à l'absorption. Pour cela, intéressons-nous à la série harmonique.

Q8. Le terme général u_n de la série harmonique est défini par $u_n = \frac{1}{n}$. La $n-ième$ somme partielle de la série harmonique a pour expression :

$$S_N = \sum_{k=1}^{N}\frac{1}{k}$$

Pour les deux questions suivantes, vous compléterez le tableau ci-dessous.

Q9. Ecrire le code python qui retourne le résultat de la somme pour des indices croissants $k = 1, \dots, N$.

Q10. Modifier le code précédent de manière à réaliser la sommation pour des indices décroissants $k = N, \dots, 1$.

	$N = 10^5$	$N = 10^6$	$N = 10^7$	$N = 10^8$
Valeur	12.090146129863428	14.392726722865724	16.695311365859852	18.997896413853898
Résultat de $1 \to N$				
Résultat de $N \to 1$				

Analyser les résultats puis conclure

Un algorithme « bien pensé » peut limiter voire compenser les erreurs. Comparativement à l'approche naïve précédente, l'algorithme de Kahan (ou sommation compensée), aussi connu sous le nom patronymique conjoint de Kahan-Babuška[2], réduit de manière significative l'erreur numérique sur la somme d'une séquence de réels représentés en virgule flottante, et d'améliorer ainsi la précision. Le pseudocode de cet algorithme est donné dans la suite.

Q11. Analyser le pseudo-code de l'algorithme de Kahan.

Q12. Décliner son code Python puis recalculer la somme pour les valeurs précédentes de N ?

Analyser les résultats et conclure.

[2] A. Klein, Kassel, « A Generalized Kahan-Babuška-Summation-Algorithm », Computing 76, 279–293 (2006)

ALGORITHME

Sommation 1/k avec l'algorithme de Kahan

```
# Déclaration des variables
VARIABLES : j, N : ENTIERS
VARIABLES : s, c, y, t : REELS

DEBUT
    # Initialisation des variables
    s ← 0                                   # s : somme,
    c ← 0                                   # c : compensation des erreurs pour les bits de poids faible,
                                            # y : valeur à ajouter à chaque itération,
                                            # t : somme temporaire.
    # Sommation avec compensation
    POUR j allant de 1 à N faire
        y ← 1/j - c                         # s grand, y petit : les bits de poids faibles de y sont perdus....
        t ← s + y                           # (t-s) donne la valeur « assimilée » de y dans l'opération
        c ←(t – s) – y                      précédente. En soustrayant y on retrouve les décimales
        s ← t                               perdues.
    FIN                                     # on corrige à l'itération suivante
    RETOURNE s

    # Affichage du résultat
    AFFICHER "La somme vaut : ", s

FIN
```

Petit glossaire sur quelques fonctions qui permettent d'embellir vos plots, linéaire, semilog, renseigner les axes, modifier les polices, la taille des markers, … ?

Soit deux listes x et y de même dimension, ci-dessous, un exemple de lignes de commandes qui donnent de la couleur à vos résultats.

```
plt.plot(x,y,'-sr',markersize=15,linewidth=2,label='signal')      # plot linéaire
plt.semilogx(x,y,'o-r')                                           # plot semilog suivant x
plt.loglog(x,y,'o-r')                                             # plot log suivant les deux axes
# renseignement des axes
plt.title("Evolution du signal et fonction du temps",fontsize=40,color = 'blue')
plt.xlabel("temps (SD)",fontsize=40,color = 'blue')
plt.ylabel("Amplitude",fontsize=40,color = 'blue')
# modification de la police des axes
plt.xticks(fontsize=30,rotation = 45,color = 'red')
plt.yticks(fontsize=30,rotation = 45,color = 'red')
plt.grid(True)
# affichage de la légende
plt.legend(fontsize=40)
```

Références bibliographiques :

[1] J. P. Demailly, « Analyse Numérique et Equations Différentielles », edp sciences.

[2] M.J. Gander and W. Gander, « Scientific Computing - An introduction using MAPLE and MATLAB ».
ISBN 978-3-319-04325-8 (eBook)

[3] Nicolas Louvet, « Algorithmes compensés en arithmétique flottante : précision, validation, performances », Thèse soutenue le 13 mai 2016. HAL Id: tel-01315543
Lien : https://theses.hal.science/tel-01315543