Projet n°1
Méthode de calcul numérique & Limites de la machine

Groupe n°4 - Equipe n°5

Responsable : lcidere
Secrétaire : rboudjeltia
Codeurs : pgaulon, ladotevi

Résumé : Ce projet consiste à nous familliariser avec le langage de programmation Python d'une part. Puis, d'autre part, à mettre en évidence la représentation machine des nombres, puisque malgré qu'il soit simple de représenter les entiers, il en demeurre plus coplexe lorsqu'il s'agit des rationnels. Ainsi, nous allons tenter de comprendre les principes de cette approximation et ses impacts sur des opérations élémentaire telles que l'addition ou la multiplication.

1 Representation des nombres en machine

1.1 Représentation décimale réduite

Afin de comprendre l'approximation machine d'un nombre, il est nécéssaire de la simuler. Ainsi, nous avons éffectué cette simulation avec le langage PYTHON et le fait que ce dernier représente les nombres avec leurs valeurs réeelles.
De plus, l'approximation sera éffectué par la fonction **rp(x,p)** calculant x avec p chiffre significatif.
Pour atteindre nos objectifs, il fut d'abord nécéssaire de connaitre ce qu'on nommera l'ordre de x, c'est à dire la puissance de 10 du premier chiffre significatif. L'algorithme qui implémente cette fonction procède de la fçon suivante. Tout d'abord, on regard si x est inférieur à 1. Dans ce cas, on multiplie successivement x par 10 jusqu'à qu'il soit inférieur à 1, l'ordre sera le nombre de fois où nous avons multiplié par 10. On fait de meme si x est supérieur à 1 mais en divisant par 10 cette fois ci.

Ensuite, nous posons n l'ordre de x. Si la p-ème décimale est suivi d'un chiffre supérieur à 5 alors on additionne par 5

Data: x: Flottant, p: Entier
Result: La représentation décimale réduite de x sur p décimale
$n = ordre(x)$;
$num = x$;
$num = num \cdot 10^{-n+p+1}$;
if $str(x)[p-n] \geq 5$ **then**
  $num = num + 5$;
end
$num = \frac{num}{10}$;
$num = E(num)$;
$num = num \cdot 10^{n-p}$;
return num;
return num ;

Algorithm 1: Représentation décimale de x sur p décimale

Prenons un exemple qui illustre bien notre algorithme,on veut $rp(x = 3.141592, p = 5)$

- $n = 1$ et $num = 3.141592$

- $num = 314159,2$
- $str(x)[4] = 9$ donc $num = 3141592$
- $num = 314165.2$
- $num = 31416,52$
- $num = 31416$
- au final on a $\underline{3.1416}$

A noter que sur PYTHON à une fonction **str(x)** qui met x (virgule incluse) dans un tableau.

1.2 Erreurs relatives

Selon les calculs éffectués par la machine, l'approximation peut avoir des variations plus ou moins importantes. Elle est évalué à l'aide de l'erreur relative qui est définie par la formule suivante:

$\delta_\star(x,y) = \frac{|(x\star y)_{real} - (x\star y)_{machine}|}{|(x\star y)_{real}|}$ (avec $\star$ qui est la somme ou le produit)
On peut donc s'apercevoir que cette grandeur tend vers 0. Afin de mieu observer l'impact de ces calculs sur l'erreur relative nous avons tracé un exemple avec le nombre e avec une précision de 3(cf fin du document). Ainsi, nous pouvons constater que l'erreur demeurre faible et converge vers 0 pour la somme mais a tendance à diverger pour le produit malgré que les valeurs restent faibles.

1.3 Calcul de log(2)

Maintenant, nous allons tenter de calculer **log(2)** sur p décimales près. Tout d'abord tout nos calculs tournerons autour de la série harmonique alterné qui converge vers **log(2)**.
$log(2) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}$
L'implémentation de cette somme nécessite qu'une simple boucle.

Afin d'obtenir une précision nous avons sommer les termes de la Série jusqu'à ce que les p nombres après la virgule correspondent à ceux du vrai log(2) (**numpy.log(2)**).

A noter que cette malgré que cette algorithme algorithme fourni une valeur approchée de log(2) avec une erreur qui tend vers 0 quand p temps vers l'infini, il sera assez long d'avoir une approximation pour p=100 par exemple.

2 Algorithme CORDIC

Le représentation des nombres flottants sur une calculatrice est en double précision. Cela signifie qu'un nombre flottant est codé sur 64 bits. Il possède 1 bit de signe, 12 bits d'exposant, dont un pour le signe de l'exposant, et la mantisse est composée de 52 bits.
Cela a pour avantage de représenter beacoup de nombres, et dans un très large intervalle. Cependant, on ne peut pas représenter tous les nombres, cela prend de la place et la précision est limitée. Les arrondis sont dûs au changement de base et peuvent s'accumuler. L'exposant étnt aussi limité il y a

Data: p: Entier
Result: Approximation de log(2) avec p décimales
$i = 1$;
$n = 2$;
$l = 1$;
$b = log(2)$;
while $i \leq p$ **do**
  while $str(b)[i] \neq str(l)[i]$ **do**
    $l = l + \frac{(-1)^{n+1}}{n}$;
    $n = n + 1$;
  end
  $i = i + 1$;
end
return l ;

Algorithm 2: Approximation de log(2) sur p décimales

un risque d'overflow ou d'underflow.

La technique générale présentée dans les algorithmes du logarithme, de l'exponentielle, de l'arctangente et de la tangente fonctionnent sur la technique suivante :

- Précalculer des valeurs de la fonction en question, ou de son inverse,
- Effectuer des transformations simples comme l'addition, la soustraction et le décalage de bits pour réduire le paramètre x à une valeur appartenant à l'intervalle sur lequel la fonction est otpimisée, et calculer le résultat en même temps,
- Obtenir le résultat par un développement en série.

Ainsi, cette technique comporte deux étapes :

- Réduire le paramètre dans l'intervalle souhaité,
- Itérer sur toutes les valeurs du tableau pour obtenir le développement en série, en commençant par le plus grand terme.

Cette technique est efficace pour les calculatrices, puisqu'elle n'utilise que la multiplication par une puissance de 10, qui se traduit par un décalage de bits, et l'addition et la soustraction qui peuvent aussi directement s'appliquer sur le nombre binaire. Ce sont des opérations dites légères. De plus les valeurs précalculées ne sont pas nombreuses, donc la mémoire occupée est relativement faible.

Ces fonctions ont été testées sur des nombres entiers aléatoires entre 0 et 100 pour les fonctions exponentielle, tangente et arctangente, et entre 1 et 100 pour la fonction logarithme. Puis nous avons comparé les résultats de ces algorithmes à ceux obtenus sur une calculatrice et donnés par Google, pour le même paramètre, en s'assurant de leur concordance.

Un des premiers problèmes soulevés par *Numerical Recipes in C* à la page 166 est l'utilisation de séries qui convergent pour tout x. Ces séries ne convergent pas assez vite pour être utilisées numériquement. On peut notamment citer la fonction sinus ou la fonction de Bessel qui convergent pour tout x, mais si on n'a pas $k >> |x|$ leurs termes augmentent.

$$\sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1}$$
$$J_n(x) = (\frac{x}{2}^n) \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!(k+n)!}$$

Une première solution est d'utiliser le procédé δ^2 d'Aitken. Si les séries S_{n-1}, S_n et S_{n+1} sont trois sommes partielles successives, une meilleure estimation de S_n est :

$$S'_n \equiv S_{n+1} - \frac{(S_{n+1}-S_n)^2}{S_{n+1}-2S_n+S_{n-1}}.$$

Cette méthode peut être appliquée telle quelle sur une machine.
Pour les séries alternées convergentes la méthode d'Euler est plus efficace. Pour n pair, elle s'écrit :

$$\sum_{s=0}^{\infty}(-1)^s u_s = u_0 - u_1 + u_2 \cdots - u_{n-1} + \sum_{s=0}^{\infty} \frac{(-1)^s}{2^{s+1}} [\Delta^s u_n]$$

Où on a :

$$\Delta u_n \equiv u_{n+1} - u_n$$
$$\Delta^2 u_n \equiv u_{n+2} - 2u_{n+1} + u_n$$
$$\Delta^3 u_n \equiv u_{n+3} - 3u_{n+2} + 3u_{n+1} - u_n.$$

Pour les séries à termes positifs, on peut se ramener à une série alternée par la méthode de Van Wijngaarden, puis appliquer la méthode d'Euler sur cette série obtenue :

$$\sum_{r=1}^{\infty} v_r = \sum_{r=1}^{\infty}(-1)^{r-1} w_r \text{ Où : } w_r \equiv v_r + 2v_{2r} + 4v_{4r} + \ldots.$$

Un autre problème soulevé dans ce livre à la page 177 vient des opérations usuelles sur les nombres complexes. Par exemple si on multiplie deux nombres complexes, il est plus rapide d'utiliser

$$(a+ib)(c+id) = (ac-bd) + i[(a+b)(c+d) - ac - bd]$$
que
$$(a+ib)(c+id) = (ac-bd) + i(bc+ad).$$

De même, utiliser $|a+ib| = \sqrt{a^2+b^2}$ peut provoquer un overflow si a ou b est grand. Il est donc préférable d'utiliser :

$$|a+ib| =$$

$$\begin{cases} |a|\sqrt{1+(b/a)^2} & |a| \geq |b| \\ |b|\sqrt{1+(a/b)^2} & |a| < |b|. \end{cases}$$

Le dernier problème que nous allons évoquer, ainsi que sa solution, portent sur les fractions continues du type : $f(x) = b_0 + \frac{a_1}{b_1+}\frac{a_2}{b_2+}\ldots$. Le problème évoqué à la page 169 est de savoir quand s'arrêter pour une telle évaluation sur machine. Une première méthode a été mise en place par J. Wallis.
On utilise une suite $f_n = \frac{A_n}{B_n}$ avec :

$$A_{-1} \equiv 1 \text{ et } B_{-1} \equiv 0$$
$$A_0 \equiv b_0 \text{ et } B_0 \equiv 1$$
$$A_j = b_j A_{j-1} + a_j A_{j-2} \text{ et } B_j = b_j B_{j-1} + a_j B_{j-2} \text{ où } j = 1, 2, \ldots, n$$

Mais sur une machine cette méthode peut créer un risque d'overflow ou d'underflow en cas de valeurs extrêmes pour le numérateur ou le dénominateur. Deux nouveaux algorithmes ont alors été proposés, par Steed puis Lentz. Et depuis, l'algorithme de Lentz a été amélioré par Thompson et Barnett.
Voici la métode de Steed :

$$D_j = \frac{B_{j-1}}{B_j}$$
$$\Delta f_j = f_j - fj - 1 \text{ qui utilise récursivement :}$$
$$D_j = \frac{1}{(b_j + a_j D_{j-1})}$$
$$\Delta f_j = (b_j D_j - 1)\Delta f_{j-1}$$

Cela peut parfois entrainer un dénominateur proche de zéro.
La meilleur méthode est donc celle de Lentz, qui utilise :

$$C_j = \frac{A_j}{A_{j-1}} \text{ et } D_j = \frac{B_{j-1}}{B_j}$$
Et calcule f_j avec :
$$f_j = f_{j-1} C_j D_j.$$

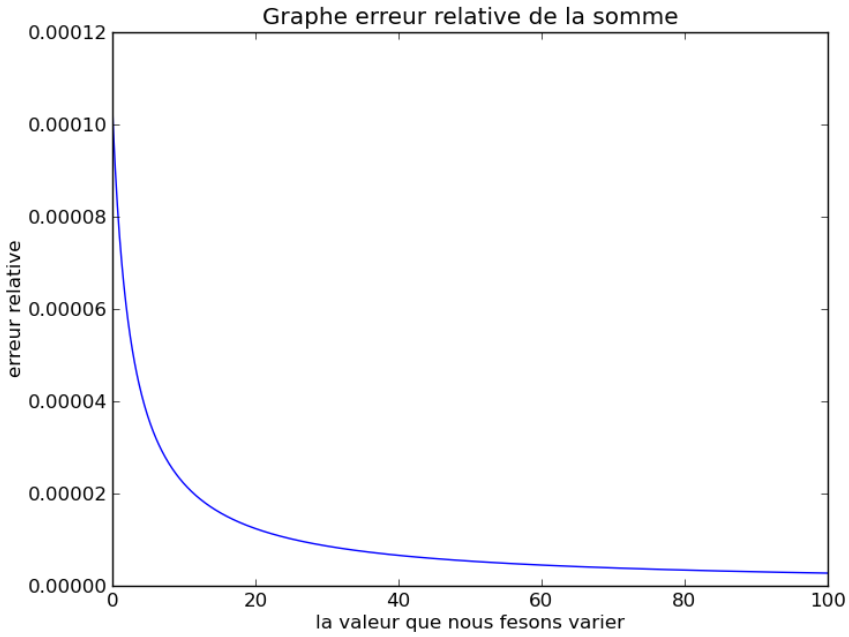
Le ratio satisfait donc la relation de récurrence :

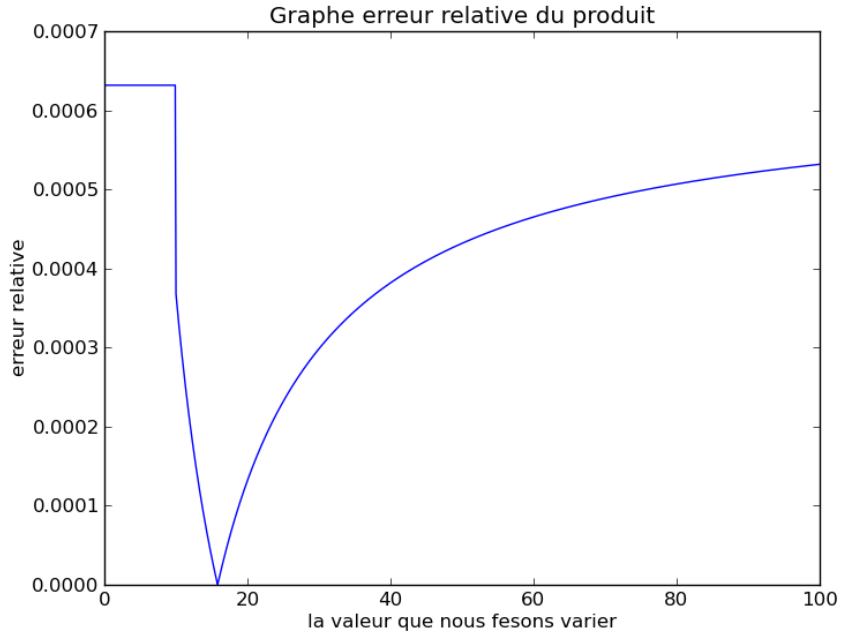
$$D_j = \frac{1}{(b_j + a_j D_{j-1})} \text{ et } C_j = b_j + a_j C_{j-1}.$$

Ici encore, le dénominateur peut approcher zéro. La modification de Thompson et Barnett impose à ce faible dénominateur une valeur telle que 10^{-30} lorsqu'il devient trop petit.

3 Conclusion

A travers d'exemple concret tel que le fonctionnement de la calculatrice, ce projet nous a permis de nous faire découvrir la réprésentation machine des nombres et la compléxité à la programmer. Ainsi, nous avons pris connaissance de méthodes utilisés qui permettent de faire des évaluations avec une précision correct pour des opérations courantes. En meme temps nous avons eu l'occasion de manipuler le langage PYTHON et deux de ces principales librairie(Numpy et Matplotlib) qui nous ont servis à tester nos algorithme.


Graphe erreur relative de la somme
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
erreur relative
0
20
40
60
80
100
la valeur que nous fesons varier


Graphe erreur relative du produit
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
erreur relative
0
20
40
60
80
100
la valeur que nous fesons varier