MTH1504 2011-2012

CALCUL SCIENTIFIQUE

Table des matières

1 Erreur absolue et erreur relative

Soit x un nombre réel, et x^*, une approximation de ce nombre. L'**erreur absolue** est définie par :

$$\Delta x = |x - x^\star|$$

L'**erreur relative** est définie par :

$$E_r(x) = \frac{|x - x^\star|}{|x|} = \frac{\Delta x}{|x|}$$

De plus, en multipliant par 100, on obtient l'**erreur relative en pourcentage**.
En pratique, il est difficile d'évaluer les erreurs absolue et relative, car on ne connaît généralement pas la valeur exacte de x et l'on n'a que $x^\star$. Dans le cas de quantités mesurées dont on ne connaît que la valeur approximative, il est impossible de calculer l'erreur absolue ; on dispose en revanche d'une borne supérieure pour cette erreur qui dépend de la précision des instruments de mesure utilisés. Cette borne est quand même appelée erreur absolue, alors qu'en fait on a :

$$|x - x^\star| \leq \Delta x,$$

ce qui peut également s'écrire :

$$x^\star - \Delta x \leq x \leq x^\star + \Delta x$$

On peut interpréter ce résultat en disant que l'on a estimé la valeur exacte de x à partir de $x^\star$ avec une incertitude de Δx de part et d'autre.
L'erreur absolue donne une mesure quantitative de l'erreur commise et l'erreur relative en mesure l'importance. Par exemple, si l'on fait usage d'un chronomètre dont la précision est de l'ordre du dixième de seconde, l'erreur absolue est bornée par $0,1$ s. Mais est-ce une erreur importante ? Dans le contexte d'un marathon d'une durée approximative de 2h20 min, l'erreur relative liée au chronométrage est très faible :

$$\frac{0,1}{2 \times 60 \times 60 + 20 \times 60} = 0,0000119$$

et on ne devrait pas avoir de conséquence sur le classement des coureurs. Par contre, s'il s'agit d'une course de 100 m d'une durée d'environ 10 s, l'erreur relative est beaucoup plus importante :

$$\frac{0,1}{10,0} = 0,01$$

soit 1% du temps de course. Avec une telle erreur, on ne pourra vraisemblablement pas distinguer le premier du dernier coureur.
Cela nous amène à parler de précision et de **chiffres significatifs**.

Définition 1.1 *Si l'erreur absolue vérifie :*

$$\Delta x \leq 0,5 \times 10^m$$

*alors le chiffre correspondant à la m-ième puissance de 10 est dit **significatif** et tous ceux à sa gauche (correspondant aux puissances de 10 supérieures à m) le sont aussi.*
Inversement, si un nombre est donné avec n chiffres significatifs, cela signifie que l'erreur absolue est inférieure à 0,5 fois la puissance de 10 correspondant au dernier chiffre significatif.

Exemple 1.1 *On obtient une approximation de π ($x = \pi$) au moyen de la quantité $\frac{22}{7}$ ($x^\star = \frac{22}{7} = 3,142857...$). On en conclut que*

$$\Delta x = \left|\pi - \frac{22}{7}\right| = 0,00126... \simeq 0,126 \times 10^{-2}$$

Puisque l'erreur absolue est plus petite que $0,5 \times 10^{-2}$, le chiffre des centièmes est significatif et on a en tout 3 chiffres significatifs ($3,14$).

Exercice 1.1 *On mesure le poids d'une personne et on trouve 90,567 kg. Sachant que l'appareil utilisé est suffisamment précis pour que tous les chiffres fournis soient significatifs, déterminez une borne pour l'erreur absolue Δx.*

Exercice 1.2 *Tous les chiffres des nombres suivants sont significatifs. Donner une borne supérieure de l'erreur absolue et estimer l'erreur relative.*

a) $0,1234$ *b)* $8,760$ *c)* $3,14156$ *d)* $0,11235 \times 10^{-3}$ *e)* $8,000$ *f)* $0,22356 \times 10^{8}$

2 Représentation des nombres sur ordinateur

Soit a un nombre réel. On peut toujours l'écrire, en le multipliant par une puissance de dix adéquate, sous la forme :

$$a = \pm 0.a_1a_2a_3\cdots \times 10^q = \pm 10^q \sum_{i=1}^{\infty} a_i \times 10^{-i}$$

où $a_i \in \{0, 1, \cdots, 9\}$ avec $a_1 \neq 0$ et $q \in \mathbb{Z}$, où $\mathbb{Z}$ est l'ensemble des nombres entiers relatifs (c'est-à-dire avec un signe). C'est ce que l'on appelle la représentation du nombre a en **virgule flottante normalisée**. q s'appelle l'**exposant**, $a_1a_2a_3\cdots$ la **mantisse** et les a_i les **chiffres** (en anglais les **digits** ou, lorsqu'on travaille dans le système binaire, les **bits** pour **binary digits**) de a. En général, un nombre réel possède une mantisse avec un nombre infini de chiffres. (π, par exemple).

Exemple 2.1 *Petit exemple en binaire pour donner une idée de comment travaille un ordinateur avec que des 0 et des 1...*

L'arithmétique de l'ordinateur est une arithmétique dérivée de l'arithmétique binaire. Pour simplifier, nous ferons comme si l'ordinateur travaillait en base 10. Dans un ordinateur, chaque nombre est placé dans un **mot** de la mémoire. Un mot est formé d'un nombre fini de cases, chacune ne pouvant contenir qu'un seul chiffre. Dans la première, on va placer le signe de a, puis les chiffres successifs de la mantisse de a, ensuite le signe de son exposant et enfin les chiffres de son exposant. Chaque mot de la mémoire d'un ordinateur ne peut donc contenir qu'un nombre fini de chiffres. Appelons t le nombre de chiffres décimaux de la mantisse des mots de l'ordinateur (en général, on aura $t = 7$ ou 8 en simple précision, 15 ou 16 en double précision). Par conséquent, seuls sont représentés de façon exacte les nombres dont la mantisse ne dépasse pas t chiffres. Pour représenter un nombre réel a ayant une mantisse avec un nombre de chiffres supérieur à t, il y a deux possibilités :

1. la **troncature** qui consiste à couper (à tronquer) la mantisse de a après son t-ième chiffre,
2. l'**arrondi** qui consiste à tenir compte du $(t+1)$-ème chiffre. Si celui-ci est inférieur à 5, on tronque, tandis que s'il est supérieur ou égal à 5, on ajoute une unité au t-ième chiffre avant de tronquer.

Presque la totalité des ordinateurs travaille actuellement par arrondi. Dans les deux cas, le nombre réel a est donc représenté dans l'ordinateur par un nombre (dit **nombre machine**) qui, en général, est une approximation. Nous l'appellerons $fl(a)$. L'erreur absolue ainsi commise s'appelle **erreur d'affectation**, parce qu'au nombre réel a on affecte un mot contenant le nombre $fl(a)$.

Théorème 2.1 *L'erreur d'affectation vérifie*

$$|a - fl(a)| \leq K\,|a|\,10^{-t}$$

où $K = 5$ si l'ordinateur travaille par arrondi et 10 s'il travaille par troncature.

Démonstration : ...

Définition 2.1 *La **précision machine** ε est la plus grande erreur relative que l'on puisse commettre en représentant un nombre réel sur ordinateur.*
Si on utilise la troncature on a donc $\varepsilon \leq 10^{1-t}$.
Si on utilise l'arrondi on a $\varepsilon \leq 0,5 \times 10^{1-t}$.

Comme l'ordinateur calcule en réalité en base 2, on aura en réalité une précision machine égale à 2^{1-24} en simple précision (23 bits), 2^{1-53} en double précision (53 bits), en supposant qu'on travaille par troncature.
Ce résultat peut être vérifié directement sur ordinateur au moyen de l'algorithme suivant :

Algorithme 2.1 *Initialisation : $\varepsilon = 1$.*
Tant que $1 + \varepsilon > 1$: effectuer $\varepsilon = \frac{\varepsilon}{2}$.
On a divisé une fois de trop : $\varepsilon = \varepsilon * 2$.
On a trouvé la précision machine ε.

Exercice 2.1 *A programmer sur machine.*

Remarque 2.1 *Dans chaque mot de la mémoire, un certain nombre de chiffres sont réservés pour l'exposant. Si l'exposant comporte trop de chiffres, il se produira ce qu'on appelle un dépassement de capacité. Pour être plus précis, on pourra parler de dépassement par défaut (**underflow** en anglais) lorsque le signe de l'exposant est négatif, et de dépassement par excès (**overflow** en anglais) s'il est positif. Le cas du dépassement par excès donne lieu, en général, à un diagnostic. Dans le cas du dépassement par défaut, certains ordinateurs donnent un diagnostic tandis que d'autres remplacent la valeur par zéro, risquant ainsi de provoquer, par la suite, une division par zéro.*

Exercice 2.2 *Chercher sur machine les plus grand et plus petit nombre utilisés pas scilab.*

3 Arithmétique flottante

Voyons maintenant comment un ordinateur s'y prend pour effectuer les quatre opérations élémentaires $+, -, \times, /$.
Soit à calculer $A = B + C$. Cette opération ne s'effectue pas dans la mémoire centrale mais dans ce que l'on appelle l'**accumulateur**. Cet accumulateur est un ensemble de trois mots dont la particularité est d'avoir des mantisses avec $2t$ chiffres au lieu de t. Pour effectuer l'opération $A = B + C$, l'ordinateur commence par recopier sans modification dans l'accumulateur celui des deux opérandes qui est le plus grand en valeur absolue. Comme, dans l'accumulateur, la mantisse doit avoir $2t$ chiffres, il la complète à droite par des zéros. Puis il recopie l'autre opérande dans l'accumulateur en faisant en sorte que son exposant soit le même que celui du premier opérande. Pour cela, on rajoute éventuellement des zéros à gauche de la mantisse. (c'est-à-dire avant le chiffre a_1).

Exemple 3.1 *Si $t = 8$, $B = 0.23487757 \times 10^3$, et $C = 0.56799442$, l'ordinateur commence par recopier B dans l'accumulateur sous la forme*

$$B = 0.2348775700000000 \times 10^3$$

Pour que C ait le même exposant que B, il faut le multiplier par 10^3, et dans l'accumulateur on a C sous la forme

$$C = 0.0005679944200000 \times 10^3$$

Maintenant, l'addition peut s'effectuer dans l'accumulateur et l'on trouve

$$B + C = 0.2354455644200000 \times 10^3$$

Enfin, notre résultat doit être envoyé dans un mot de la mémoire de l'ordinateur. Or, ces mots ont des mantisses qui ne possèdent que t (8 ici) chiffres. Nous allons donc faire une erreur qui est celle que l'on commet lorsque l'on place un nombre ayant une mantisse de plus de t chiffres dans un mot qui n'en accepte que t : c'est une erreur d'affectation. On obtient ici :

$$A = 0.23544556 \times 10^3$$

Pour chacune des quatre opérations élémentaires, l'erreur vérifie :

Théorème 3.1 *L'erreur sur une opération arithmétique satisfait*

$$|(B \star C) - fl(B \star C)| \leq K\,|B \star C|\,10^{-t}$$

avec $K = 5$ si l'ordinateur travaille par arrondi et 10 s'il travaille par troncature, et où $\star$ est l'une des quatre opérations $+, -, \times, /$.

Exercice 3.1 *Calculer, avec $t = 8$, $fl(B + C)$ dans le cas :*

$$B = 0.56543451 \times 10^6, \quad C = 0.21554623 \times 10^{-4}$$

3.1 Absorption

Considérons maintenant $B = 0.10000000 \times 10^1$ et $C = 0.30000000 \times 10^{-7}$, avec toujours $t = 8$. Dans l'accumulateur, on aura

$$C = 0.000000003000000 \times 10^1, \quad B + C = 0.1000000030000000 \times 10^1$$

En revenant dans la mémoire, on obtiendra $A = 0.10000000 \times 10^1$, c'est-à-dire que $A = B$. Le petit nombre C a complètement disparu devant le plus grand : on dit qu'il y a **absorption** de C par B. Donc, sur ordinateur,

$$fl(1 + \alpha) = 1 \text{ pour } \alpha \text{ suffisamment petit}$$

Le plus grand des nombres α tels que $fl(1 + \alpha) = 1$ n'est autre que la précision machine : on a $fl(1 + \varepsilon) > 1$ et pour tout $\alpha < \varepsilon$, $fl(1 + \alpha) = 1$.
Conclusion : on peut vérifier que

$$\frac{C}{B} < \varepsilon \Rightarrow fl(B + C) = B$$

Une conséquence importante de l'erreur d'absorption est la non associativité de l'addition machine. Considérons les deux quantités

$$v = y + (x - x)$$
$$u = (y + x) - x$$

Un ordinateur ne peut effectuer qu'une seule opération arithmétique à la fois. Les parenthèses dans les expressions précédentes indiquent laquelle des deux opérations doit être faite en premier. Supposons que $x = 1$ et que $y = \alpha$ avec $fl(1 + \alpha) = 1$. On trouve que $fl(v) = \alpha$ ce qui est la bonne réponse, mais par contre on a $fl(u) = 0$.
Conclusion : : sur ordinateur, l'addition n'est pas associative
On pourrait penser, en voyant notre exemple précédent, que de telles erreurs ne sont pas très importantes. Effectuons maintenant les calculs suivants :

$$v = \frac{y + (x - x)}{y} = 1$$
$$u = \frac{(y + x) - x}{y} = 1$$

pour $y \neq 0$. Prenons de nouveau $x = 1$ et $y = \alpha$ avec $fl(1 + \alpha) = 1$. On obtient $fl(v) = 1$, qui est la réponse correcte, mais $fl(u) = 0$. On ne peut plus dire maintenant que l'erreur soit faible !

3.2 Cancellation

Une erreur de **cancellation** se produit généralement lorsqu'on soustrait deux quantités de même signe très proches.

Exemple 3.2

$$fl(0,5678 \times 10^6 - 0,5677 \times 10^6) = fl(0,1000 \times 10^6) = 0,1 \times 10^3$$

La soustraction de ces deux nombres de valeur très proche fait apparaître trois 0 non significatifs dans la mantisse du résultat. Il y a eu une perte de chiffres significatifs.
Il faut donc éviter au maximum la possibilité que des erreurs de cancellation se produisent. Par exemple, on n'utilisera pas la formule $\frac{1}{x} - \frac{1}{x+1}$ mais on la remplacera par $\frac{1}{x(x+1)}$.

4 Conditionnement et stabilité

4.1 Conditionnement

Lorsque l'on veut effectuer un calcul, la première opération consiste à introduire des données dans l'ordinateur. Or, d'après ce que nous avons dit plus haut, certaines données peuvent être entachées d'une erreur d'affectation (par exemple, $\frac{1}{3}$ n'est pas représenté exactement en machine). Cela signifie que le problème que l'on va résoudre dans l'ordinateur n'est pas exactement celui que l'on voulait traiter. Or, il se peut que la solution exacte de ce problème perturbé soit très éloignée de la solution exacte du problème non perturbé. On voit que cette notion est afférente au problème lui-même et est indépendante de l'algorithme que l'on va utiliser pour le résoudre, ainsi que de la propagation, dans cet algorithme, des erreurs dues à l'arithmétique de l'ordinateur. Le problème mathématique peut être plus ou moins sensible à de petites variations sur les données. Ce phénomène s'appelle le **conditionnement** du problème. On dit qu'un problème est **bien conditionné** si une "petite" variation des données n'entraîne qu'une "petite" variation des résultats. Inversement, on dit qu'un problème est **mal conditionné** si une "petite" variation des données peut entraîner une "grande" variation des résultats.

Exemple 4.1 *On considère le système d'équations linéaires :*

$$\begin{cases} x - 1,99995y = 3 \\ 2x - 4y = 1 \end{cases} \tag{1}$$

On obtient la solution

$$\begin{cases} x = 100000 \\ y = 50000 \end{cases}$$

On considère le système perturbé :

$$\begin{cases} x - 1,9999y = 3 \\ 2x - 4y = 1 \end{cases} \tag{2}$$

L'erreur relative commise sur les données est

$$\frac{|1,99995 - 1,9999|}{1,99995} \simeq 2,5 \times 10^{-5}$$

La solution de (2) est donnée par :

$$\begin{cases} x = 50001 \\ y = 25000 \end{cases}$$

On observe une erreur relative d'environ 50% sur chacune des solutions x, y. Le système est ici mal conditionné.

4.2 Stabilité

On utilise, pour effectuer les calculs, un certain algorithme. Les erreurs dues à l'arithmétique de l'ordinateur peuvent se propager plus ou moins dans cet algorithme. On voit que cette notion est attachée à l'algorithme utilisé et non pas au problème mathématique traité. Cette notion s'appelle **stabilité numérique** de l'algorithme.

4.2.1 Exemple : résolution de $x^2 - 160x + 1 = 0$

Soit à résoudre l'équation

$$x^2 - 160x + 1 = 0$$

en effectuant les calculs dans une arithmétique flottante normalisée à $t = 5$ digits. Si on utilise les formules classiques donnant les solutions d'une équation du second degré, l'algorithme s'écrit :

Algorithme 4.1
Initialisation : données $a = 1$, $b = -160$, $c = 1$
$delta = b^2 - 4ac$; $racDelta = \sqrt{delta}$
$x1 = \dfrac{-b + racDelta}{2a}$; $x2 = \dfrac{-b - racDelta}{2a}$

$$delta = 25596, racDelta = 159,9875$$

et on obtient : $fl(racDelta) = 0.15999 \times 10^3$

$$fl(x1) = 160$$

$$fl(x2) = 0.5 \times 10^{-2}$$

Dans le calcul de la plus petite racine x_2, une erreur de cancellation a été commise. En conservant la valeur la plus fiable x_1 et en utilisant la formule $x_2 x_1 = \frac{c}{a}$, on obtient pour x_2 :

$$fl(x2) = 0.6250 \times 10^{-2},$$

qui est beaucoup plus proche de la valeur exacte $x_2 = 0.6252 \times 10^{-2}$.
On constate que l'algorithme 4.1 est instable, en raison de la formule donnant $x1$, qui risque de produire une erreur de cancellation. On modifie donc cet algorithme en remplaçant la formule de calcul de $x1$ par la formule : $x1 = \dfrac{c}{ax_2}$.

Exercice 4.1 *A tester sur machine.*

4.2.2 Exemple : calcul approché de e^{-10}

Supposons qu'on utilise pour cela la série :

$$e^{-10} \simeq \sum_{k=0}^{n} (-1)^k \frac{10^k}{k!},$$

les calculs étant toujours effectués avec dix chiffres significatifs. Le terme général $u_k = (-1)^k \frac{10^k}{k!}$ est tel que

$$\frac{|u_k|}{|u_{k-1}|} = \frac{10}{k} \geq 1 \text{ dès que } k \leq 10$$

On a donc deux termes de valeur absolue maximale

$$|u_9| = |u_{10}| = \frac{10^{10}}{10!} \simeq 2,755 \times 10^3$$

tandis que $e^{-10} \simeq 4,5 \times 10^{-5}$. Comparons u_{10} et e^{-10} :

$$\begin{aligned} u_{10} = &\quad 2755 \quad ,000000 \\ e^{-10} = &\quad\quad 0 \quad ,000045 \end{aligned}$$

Ceci signifie qu'au moins 8 chiffres significatifs vont être perdus par compensation des termes u_k de signes opposés. Un remède simple consiste à utiliser la relation

$$e^{-10} = \frac{1}{e^{10}} \text{ avec } e^{10} \simeq \sum_{k=0}^{n} \frac{10^k}{k!}$$

Exercice 4.2 *A tester sur machine.*

5 Coût de calcul

En général, un problème est résolu sur un ordinateur à l'aide d'un algorithme, qui est une procédure se présentant sous la forme d'un texte qui spécifie l'exécution d'une séquence finie d'opérations élémentaires.

5.1 Nombre d'opérations

Le **coût de calcul** d'un algorithme est le nombre d'opérations en virgule flottante requises pour son exécution. On mesure souvent la vitesse d'un ordinateur par le nombre maximum d'opérations en virgule flottante qu'il peut effectuer en une seconde (en abrégé **flops**). Les abréviations suivantes sont couramment utilisées : Mega-flops pour 10^6 flops, Gigaflops pour 10^9 flops, Tera-flops pour 10^{12} flops. Les ordinateurs actuels les plus rapides atteignent 40 Tera-flops.
En général, il n'est pas essentiel de connaître le nombre exact d'opérations effectuées par un algorithme. Il est suffisant de se contenter de l'ordre de grandeur en fonction d'un paramètre d relié à la dimension du problème. On dit qu'un algorithme a une complexité constante s'il requiert un nombre d'opérations indépendant de d, i.e. $O(1)$ opérations. On dit qu'il a une complexité linéaire s'il requiert $O(d)$ opérations, ou, plus généralement, une complexité polynomiale s'il requiert $O(d^m)$ opérations, où m est un entier positif. Des algorithmes peuvent aussi avoir une complexité exponentielle ($O(c^d)$ opérations) ou même factorielle ($O(d!)$ opérations). Rappelons que l'écriture $O(d^m)$ signifie : "se comporte pour de grandes valeurs de d, comme une constante fois d^m".

Exemple 5.1 ***Produit matrice-vecteur***
Soit A une matrice carrée d'ordre n et soit $v \in \mathbb{R}^n$. La j-ième composante du produit Av est donnée par :

$$a_{j1}v_1 + a_{j2}v_2 + \cdots a_{jn}v_n,$$

ce qui nécessite n produits et $n-1$ additions. On effectue donc $n(2n-1)$ opérations pour calculer toutes les composantes. Cet algorithme requiert $O(n^2)$ opérations, il a donc une complexité quadratique par rapport au paramètre n. Le même algorithme nécessiterait $O(n^3)$ opérations pour calculer le produit de deux matrices d'ordre n. Il existe un autre algorithme, dû à Strassen, qui ne requiert "que" $O(n^{\log_2^7})$ opérations.

Exercice 5.1 *Quelle est la complexité d'un algorithme de calcul du déterminant qui utilise la formule de développement par rapport à une ligne ? Rappel :*

$$\det(A) = \sum_{j=1}^{n} \Delta_{ij} a_{ij}$$

Montrer que si $n = 24$, il faudrait 20 ans à un ordinateur capable d'atteindre 1 Peta-flops (i.e. 10^{15} opérations par seconde).

5.2 Temps de calcul

Le nombre d'opérations n'est pas le seul paramètre à prendre en compte dans l'analyse d'un algorithme. Un autre facteur important est le temps d'accès à la mémoire de l'ordinateur (qui dépend de la manière dont l'algorithme a été programmé). Un indicateur de la performance d'un algorithme est donc le temps CPU (Central Processing Unit), c'est-à-dire le temps de calcul (commande `timer()` en scilab).

Exercice 5.2 *Cet exercice a pour but de comparer différentes définitions de matrices. On se servira de la fonction* ***`timer`*** *de Scilab qui permet de mesurer le temps mis par Scilab à effectuer certaines opérations. L'appel* ***`timer()`*** *renvoie le temps écoulé depuis le dernier appel* ***`timer()`****.*
Exécuter les instructions suivantes :

```
n=400;
timer();
for j=1: n, for i=1: n, a(i,j)=cos(i)*sin(j);end;end;
t1=timer();clear a;

timer();
a=zeros(n,n);
for j=1: n, for i=1: n, a(i,j)=cos(i)*sin(j);end;end;
t2=timer();clear a;

timer();
a=zeros(n,n);
for i=1: n, for j=1: n, a(i,j)=cos(i)*sin(j);end;end;
t3=timer();clear a;

timer();
a=zeros(n,n);
a=cos(1:n)'*sin(1:n);
t4=timer();clear a;
```