Chapitre 1

Généralités sur l'analyse numérique et le calcul scientifique

1.1 Motivation

L'analyse numérique (numerical analysis en anglais) est une branche des mathématiques appliquées s'intéressant au développement d'outils et de méthodes numériques pour le calcul d'approximations de solutions de problèmes de mathématiques qu'il serait difficile, voire impossible, d'obtenir par des moyens analytiques. Son objectif est notamment d'introduire des procédures calculatoires détaillées susceptibles d'être mises en œuvre par des calculateurs (électroniques, mécaniques ou humains) et d'analyser leurs caractéristiques et leurs performances. Elle possède des liens étroits avec deux disciplines à la croisée des mathématiques et de l'informatique. La première est l'analyse des algorithmes (analysis of algorithmsen anglais), elle-même une branche de lathéorie de la complexité (computational complexity theoryen anglais), qui fournit une mesure de l'efficacité d'une méthode en quantifiant le nombre d'opérations élémentaires, ou parfois la quantité de ressources informatiques (comme le temps de calcul, le besoin en mémoire...), qu'elle requiert pour la résolution d'un problème donné. La seconde est le calcul scientifique (scientific computingen anglais), qui consiste en l'étude de l'implémentation de méthodes

numériques dans des architectures d'ordinateurs et leur application à la résolution effective de problèmes issus de la physique, de la biologie, des sciences de l'ingénieur ou encore de l'économie et de la finance.

1.2 Arithmétique en virgule flottante et erreurs d'arrondis

La mise en œuvre d'une méthode numérique sur une machine amène un certain nombre de difficultés d'ordre pratique, qui sont principalement liées à la nécessaire représentation approchée des nombres réels en mémoire.

Avant de décrire plusieurs des particularités de l'arithmétique à virgule flottante (floating-point arithmetic en anglais) en usage sur la majorité des ordinateurs et calculateurs actuels, les principes de représentation des nombres réels et de leur stockage en machine sont rappelés.

1.2.1 Représentation des nombres en machine

La mémoire d'une machine étant constituée d'un support physique, sa capacité est, par construction, limitée. Pour cette raison, le nombre de valeurs (entières, réelles, etc.) représentables, stockées en machine sous la forme d'ensembles de chiffres affectés à des cellules-mémoire portant le nom de mots-mémoire, est fini. Pour les nombres réels, il existe essentiellement deux systèmes de représentation : celui des nombres à virgule fixe et celui des nombres à virgule flottante.

Représentation des réels en virgule fixe

Les nombres réels sont les éléments d'un corps archimédien complet totalement ordonné noté $\mathbb{R}$, constitué de nombres dits rationnels, comme 76 ou $\frac{-4}{3}$, et de nombres dits irrationnels, comme $\sqrt{2}$ ou π. On peut les représenter grâce à un système de numération positionnel relatif au choix d'une base (base ou encore radix en anglais) β, $\beta \in \mathbb{N}$, $\beta \geq 2$,

en utilisant que:

$$x \in \mathbb{R} \Longleftrightarrow x = \pm \left(\underbrace{\sum_{i=0}^{i=p} x_i \beta^i}_{y} + \underbrace{\sum_{i=1}^{i=+\infty} x_{-i} \beta^{-i}}_{z} \right),$$

où $p \in \mathbb{N}$ et les coefficients x_i,x_{-i}, prennent leurs valeurs dans l'ensemble $\{0, ..., \beta - 1\}$. On écrit alors conventionnellement:

$$x = \left(\pm x_p x_{p-1} ... x_0, x_{-1} ... x_{-n} ... \right)_\beta ,$$

où la virgule est le séparateur entre la partie entière $y = E(x)$ et la partie fractionnaire $z \in [0; 1]$ du réel x, l'indice final précisant simplement que la représentation du nombre est faite relativement à la base β, appellée la représentation en virgule fixe de réel x.

Les bases utilisées couramment sont:

$\beta = 2$ base binaire, les coefficients b_i peuvent prendre les valeurs 0 et 1.

$\beta = 8$ base octale, les coefficients b_i peuvent prendre les valeurs $0, 1, 2, ..., 7$.

$\beta = 10$ base décimale, les coefficients b_i sont pris dans l'ensemble $\{0, 1, 2, ..., 9\}$.

$\beta = 16$ base hexadécimale, les coefficients $b_i \in \{0, 1, 2, ..., 9, A = 10, B = 11, C = 12, D = 13, E = 1$

Exemple 1.2.1 $(10110)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (22)_{10}$.

$(3AF)_{16} = 3 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = (943)_{10}$.

$(10011,011)_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = (19,375)_{10}$.

$(11)_{10} = (1011)_2$.

$$\begin{array}{cccccc} 11 & 2 & & & \\ 1 & 5 & 2 & & \\ \nwarrow & 1 & 2 & 2 & \\ & \nwarrow & 0 & 1 & 2 \\ & & \nwarrow & 1 & 0 \end{array}$$

$(92)_{10} = (5C)_{16}$.

$$\begin{array}{llll} 92 & 16 & & \\ 12 & 5 & 16 & \\ & 5 & 0 & \end{array}$$

$(2,75)_{10} = 2 + \frac{1}{2} + \frac{1}{4} = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = (10,11)_2$.

$(0,375)_{10} = (0,011)_2$.

$$\begin{array}{lll} z_i & \beta z_i & E(\beta z_i) \\ 0,375 & 0,375 & 0 \\ 0,750 & 1,5 & 1 \\ 0,5 & 1 & 1 \\ 0 & & \end{array}$$

$\left(\frac{1}{3}\right)_{10} = (0,333...)_{10} = \left(0,\overline{3}\right)_{10} = (0,010101...)_2 = \left(0,\overline{01}\right)_2$.

$$\begin{array}{lll} z_i & \beta z_i & E(\beta z_i) \\ \frac{1}{3} & \frac{2}{3} & 0 \\ \frac{2}{3} & \frac{4}{3} & 1 \\ \frac{1}{3} & \frac{2}{3} & 0 \\ ... & ... & ... \end{array}$$

$\left(\frac{1}{3}\right)_{10} = (0,333...)_{10} = \left(0,\overline{3}\right)_{10} = (0,252525...)_8 = \left(0,\overline{25}\right)_8$.

$$\begin{array}{lll} z_i & \beta z_i & E(\beta z_i) \\ \frac{1}{3} & \frac{8}{3} & 2 \\ \frac{2}{3} & \frac{16}{3} & 5 \\ \frac{1}{3} & \frac{8}{3} & 2 \\ ... & ... & ... \end{array}$$

$\left(0,\overline{0011}\right)_2 = (0,2)_{10}$.

$$\begin{aligned} \left(0,\overline{0011}\right)_2 &= 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + ... \\ &= \sum_{i=1}^{+\infty} \left[2^{-4i} + 2^{-4i+1}\right] = 3\sum_{i=1}^{+\infty} 2^{-4i} \\ &= \frac{3}{16}\sum_{i=0}^{+\infty} \left(\frac{1}{16}\right)^i = \frac{1}{5} = (0,2)_{10} . \end{aligned}$$

Le dernier des exemples ci-dessus montre qu'à la représentation finie d'un nombre réel dans le système décimal peut parfaitement correspondre une représentation infinie non triviale dans le système binaire.

Représentation des nombres réels en virgule flottante

Soit $\beta \geq 2$ une base. tout réel x non nul peut s'écrire sous la forme:

$$\begin{aligned} x &= \pm\beta^e \sum_{i=1}^{+\infty} x_{-i}\beta^{-i} \\ &= \pm\beta^e \times m \\ &= \pm\beta^e \times (0, x_{-1}...x_{-n}...)_\beta \,, \text{ avec } x_{-1} \neq 0. \end{aligned}$$

e s'appelle l'exposant et m la mantisse. Les chiffres $x_{-1}, x_{-2}, ..., x_{-n}, ...$ sont les chiffres significatifs de x en base β.

Cette écriture est appelée représentation en virgule flottante normalisée (VFN) du réel x.

On écrira aussi en décalant la virgule d'une position vers la droite:

$$x = \pm\beta^{e-1} \times (x_{-1}, x_{-2}...x_{-n}...)_\beta \,.$$

C'est l'écriture scientifique d'un réel.

Exemple 1.2.2 *Le réel* $x = 61,123$ *en base* 10 *s'écrit*

$$x = (61,123)_{10} = 10^2 \times (0,61123)_{10} = 10^1 \times (6,1123)_{10} \,.$$

Le réel $x = 1001,1011$ *en base* 2 *s'écrit*

$$x = (1001,1011)_2 = 2^4 \times (0,10011011)_2 = 2^3 \times (1,0011011)_2 \,.$$

1.2.2 Erreurs d'arrondis

Une machine (ordinateur, calculatrice,...) ne peut stocker qu'un nombre fini de chiffres pour représenter un réel donné. On utilise pour cele la représentation en virgule flottante normalisée, en arrondissant ou en tronquant la mantisse à t chiffres.

Les réels en virgule flottante normalisée à t chiffres

Trancature et arrondi en base 10: Soit x un réel quelconque (en V.F.N)

$$x = \pm 10^e \times (0, x_{-1}x_{-2}...x_{-n}...)_{10}\,.$$

Alors sa représentation en virgule flottante normalisée à t chiffres en **troncature** est le nombre:

$$\widehat{x} = \pm 10^e \times (0, x_{-1}x_{-2}...x_{-t})_{10}\,.$$

Sa représentation machine en virgule flottante normalisée à t chiffres en **arrondi** est le nombre:

$$fl\,(x) = \pm 10^e \times m.$$

où

$$m = \begin{cases} 0, x_{-1}x_{-2}...x_{-t}, & \text{si } x_{-t-1} < 5 \\ 0, x_{-1}x_{-2}...x_{-t} + 10^{-t}, & \text{si } x_{-t-1} \geq 5 \end{cases}$$

Dans le deuxième cas, on renormalisera si nécessaire l'écriture de $fl\,(x)$.

L'arrondi est en fait le nombre le plus proche de x dont la mantisse possède t chiffres.

Exemple 1.2.3 *soit $t = 3$. Alors, en arrondi*

$$\begin{aligned} x &= (1976)_{10} \quad ; \quad fl(x) = 10^4 \times (0,198)_{10}\,, \\ x &= (1962)_{10} \quad ; \quad fl(x) = 10^4 \times (0,196)_{10}\,, \\ x &= (1435)_{10} \quad ; \quad fl(x) = 10^4 \times (0,144)_{10}\,, \\ x &= (1995)_{10} \quad ; \quad fl(x) = 10^4 \times (0,2)_{10} \end{aligned}$$

En troncature

$$\begin{aligned} x &= (1976)_{10} \quad ; \quad \widehat{x} = 10^4 \times (0,197)_{10}\,, \\ x &= (1962)_{10} \quad ; \quad \widehat{x} = 10^4 \times (0,196)_{10}\,, \end{aligned}$$

Trancature et arrondi en base β: Soit x un réel quelconque (en V.F.N)

$$x = \pm\beta^e \times (0, x_{-1}x_{-2}...x_{-n}...)_\beta .$$

Alors sa représentation en virgule flottante normalisée à t chiffres en **troncature** est le nombre:

$$x = \pm\beta^e \times (0, x_{-1}x_{-2}...x_{-t})_\beta .$$

Sa représentation machine en virgule flottante normalisée à t chiffres en **arrondi** est le nombre:

$$fl(x) = \pm\beta^e \times m.$$

où

$$m = \begin{cases} 0, x_{-1}x_{-2}...x_{-t}, & \text{si } x_{-t-1} < \frac{\beta}{2} \\ 0, x_{-1}x_{-2}...x_{-t} + \beta^{-t}, & \text{si } x_{-t-1} \geq \frac{\beta}{2} \end{cases}$$

Dans le deuxième cas, on renormalisera si nécessaire l'écriture de $fl(x)$.

Exemple 1.2.4 *soit $t = 4$. Alors, en arrondi*

$$\begin{aligned} x &= (10111)_2 = 2^5 \times (0,10111)_2 \quad ; \quad fl(x) = 2^5 \times (0,1100)_2 , \\ x &= (1111001111)_2 = 2^{10} \times (0,1111001111)_2 \quad ; \quad fl(x) = 2^{10} \times (0,1111)_2 , \\ x &= (0,0001011001)_2 = 2^{-3} \times (0,1011001)_2 \quad ; \quad fl(x) = 2^{-3} \times (0,1011)_2 , \end{aligned}$$

En troncature

$$\begin{aligned} x &= (10111)_2 = 2^5 \times (0,10111)_2 \quad ; \quad \widehat{x} = 2^5 \times (0,1011)_2 , \\ x &= (1111001111)_2 = 2^{10} \times (0,1111001111)_2 \quad ; \quad \widehat{x} = 2^{10} \times (0,1111)_2 , \\ x &= (0,0001011001)_2 = 2^{-3} \times (0,1011001)_2 \quad ; \quad \widehat{x} = 2^{-3} \times (0,1011)_2 , \end{aligned}$$

Les formats machine float et double En pratique, la plupart des ordinateurs utilise la base 2. La norme actuelle retient deux formats standards de représentation des réels :

Le format double (flottants en double précision)

Pour un réel x non nul donné, on considère son écriture en virgule flottante normalisée arrondie à 53 chiffres

$$fl\,(x) = \pm 2^e \times (a_0, a_{-1}a_{-2}...a_{-t})_2,$$

avec $a_0 \neq 0$, $t = 52$.

La mantisse a son premier chiffre a_0 différent de 0, donc nécessairement égal à 1 en base 2 et il n'est pas nécessaire de le représenter en machine. C'est la convention du "bit implicite". On ne mémorisera donc que la partie fractionnaire de la mantisse

$$f = a_{-1}a_{-2}...a_{-52}.$$

Pour représenter l'exposant en base 2 avec 11 chiffres, on impose

$$-2^{10} + 2 = -1022 \leq e \leq 1023 = 2^{10} - 1,$$

et on représentera l'exposant biaisé

$$e_b = 1023 + e$$

compris entre 1 et 2046. (Les exposants biaisés 0 et 2047 sont utilisés pour représenter des nombres dénormalisés, notamment 0 et inf).

Enfin le signe sera mémorisé à l'aide d'un bit s égal à 1 si le nombre x est négatif, égal à 0 si x est positif.

Avec ces notations, le nombre x sera représenté sur 64 bits répartis ainsi:

Le format float (flottants en simple précision)

Pour ce format, les réels sont stockés sur 32 bits, 1 pour le signe, 8 pour l'exposant et 23 pour la mantisse, suivant le même principe.

Valeurs approchées et type d'erreurs

Définition 1.2.1 *On appelle valeur approchée d'un nombre réel x tout nombre réel $\tilde{x}$ voisin de x, on écrit $\tilde{x} \simeq x$ ou $\tilde{x} \approx x$.*

Si $\tilde{x} > x$, $\tilde{x}$ est dite valeur approchée par excès.

Si $\tilde{x} < x$, $\tilde{x}$ est dite valeur approchée par défaut.

Exemple 1.2.5 *$\frac{2}{3} \simeq 0,6666$, $\pi \simeq 3,14$,$\sqrt{5} \simeq 2,23$, sont des approximations par défaut mais $e \simeq 2,72$ est une approximation par excès.*

Définition 1.2.2 *L'erreur associée à la valeur approchée $\tilde{x}$ du réel x est le réel $\Delta x = x - \tilde{x}$.*

L'erreur absolue associée à la valeur approchée $\tilde{x}$ du réel x est le réel $\Delta x = |x - \tilde{x}|$.

L'erreur relative associée à la valeur approchée $\tilde{x}$ du réel x est le réel $\delta x = \frac{|x-\tilde{x}|}{|x|} = \frac{\Delta x}{|x|}$.

Remarque 1.2.1 *δx est souvent exprimé en pourcentage.*

Exemple 1.2.6 *pour $x = 1,25$, on considère la valeur approchée $\tilde{x} = 1,2$. Alors:*
$\Delta x = |x - \tilde{x}| = 0,05$ *et* $\delta x = \frac{\Delta x}{|x|} = \frac{0,05}{1,25} = 0,04 = 4 \times 10^{-2} = 4\%$.

Erreur d'arrondi Etant donné un nombre réel x, si on note $fl(x)$ sa représentation machine en virgule flottante normalisée à t chiffres en **arrondi** dans une base donnée, on appelle erreur d'arrondi le nombre

$$\Delta x = |x - fl(x)|$$

Exemple 1.2.7 *Soit $x = \frac{110}{3}$, la valeur arrondi à $t = 5$ chiffres en base 10 de x est:*

$$x = \frac{110}{3} = 10^2 \times (0,366666...)_{10}.$$

On a $fl(x) = 10^2 \times (0,36667)_{10}$, et $\Delta x = |x - fl(x)| = 10^2 \times (0,0000033...)_{10}$.

Majoration des erreurs: Si la valeur exacte n'est pas connue donc les erreurs deviennent inconnues et pour les précisées on introduit la notion de majoration des erreurs.

Définition 1.2.3 *On appelle majoration de l'erreur absolue d'une va leur approchée $\tilde{x}$, tout réel positif Δx vérifiant: $|x - \tilde{x}| \leq \Delta x$.*

Il résulte bien-sûr de ces définitions que l'on a alors $x \simeq \widetilde{x} \pm \Delta x$ et $x \simeq \widetilde{x}\,(1 \pm \delta x)$.

Exemple 1.2.8 *Dans l'exemple ci-dessus, on obtient une majoration de l'erreur,* $|x - fl\,(x)| = 10^2 \times (0,0000033...)_{10} \leq 10^2 \times (0,000004)_{10} = 10^2 \times 4 \times 10^{-6} = 4 \times 10^{-4}$.

On obtient aussi une majoration de l'erreur relative en remarquant que: $\frac{1}{|x|} = \frac{1}{10^2 \times (0,366666...)_{10}} \leq \frac{1}{10^2 \times (0,3)_{10}} = \frac{1}{10^2 \times 3 \times 10^{-1}}$,
d'où $\frac{|x - fl(x)|}{|x|} \leq \frac{10^2 \times 4 \times 10^{-6}}{10^2 \times 4 \times 10^{-1}} = \frac{4}{3} \times 10^{-5}$.

Théorème 1.2.1 *Soit le réel positif* $x = \beta^e \times (0, x_{-1}x_{-2}...x_{-n}...)_\beta$
et $fl\,(x)$ *les nombres en virgule flottante normalisée à* t *chiffres et en arrondi, on a :*

- $|x - fl\,(x)| \leq \frac{1}{2}\,|x| \times 10^{-t+1}$ *(en base* $\beta = 10$*).*
- $|x - fl\,(x)| \leq \frac{1}{2}\,|x| \times 2^{-t+1}$ *(en base* $\beta = 2$*).*

Le théorème ci-dessus permet d'obtenir, en base β quelconque, la majoration

$$\frac{|x - fl\,(x)|}{|x|} \leq \frac{1}{2} \times \beta^{-t+1} = \varepsilon_{mach}$$

Ce nombre ε_{mach} s'appelle "epsilon machine".

Remarque 1.2.2 *L'erreur relative due à l'approximation d'un nombre par sa représentation binaire en machine est :*

- $\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} \times 2^{-53+1} = 2^{-53} \simeq 1,1 \times 10^{-16}$ *(au format double).*
- $\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} \times 2^{-24+1} = 2^{-24} \simeq 6 \times 10^{-8}$ *(au format float).*

  On retiendra en particulier pour les calculs numériques avec Matlab, qui se font au format double, que

$$\varepsilon_{mach} = 2^{-53}.$$

On pourra vérifier que ce nombre $2 \times \varepsilon_{mach}$ *est égal à la constante eps de Matlab .*

Propagation des erreurs par les opérations arithmétiques: Soit a et b deux réel et Δa, Δb, δa, δb les erreurs absolues et relatives de a et de b respectivement.

Opération	Erreur absolue	Erreur relative
$a \pm b$	$\Delta a + \Delta b$	$\frac{\Delta a + \Delta b}{\lvert a \pm b \rvert}$
$a \times b$	$\lvert b \rvert \Delta a + \lvert a \rvert \Delta b$	$\frac{\Delta a}{\lvert a \rvert} + \frac{\Delta b}{\lvert b \rvert}$
$\frac{a}{b}$	$\frac{\lvert b \rvert \Delta a + \lvert a \rvert \Delta b}{b^2}$	$\frac{\Delta a}{\lvert a \rvert} + \frac{\Delta b}{\lvert b \rvert}$

1.3 Conditionnement d'un problème et Stabilité des méthodes numériques

1.3.1 Problème bien posé:

Définition 1.3.1 *Considérons le problème suivant: connaissant d, trouver x tel que*

$$F(d, x) = 0,$$

où F désigne une relation fonctionnelle liant la solution x à la donnée d du problème, ces dernières variables étant supposées appartenir à des espaces vectoriels normés sur $\mathbb{R}$ ou $\mathbb{C}$.

Un tel problème est dit bien posé au sens de Hadamard si, pour une donnée d fixée, une solution x existe, qu'elle est unique et qu'elle dépend continûment de la donnée d.

La première de ces conditions semble être la moindre des choses à exiger du problème que l'on cherche à résoudre : il faut qu'il admette au moins une solution. La seconde condition exclut de la définition les problèmes possédant plusieurs, voire une infinité de, solutions, car une telle multiplicité cache une indétermination du modèle sur lequel est basé le problème. La dernière condition, qui est la moins évidentea priori, est absolument fondamentale dans la perspective de l'utilisation de méthodes numériques de résolution. En effet, si de petites incertitudes sur les données peuvent conduire à de grandes variations sur la solution, il sera quasiment impossible d'obtenir une approximation valable de cette dernière par un calcul dans une arithmétique en précision finie.

1.3.2 Conditionnement d'un problème et stabilité des méthodes numériques:

Le fait que certains nombres ne soient pas représentés de façon exacte dans un ordinateur entraine que l'introduction même de donnée d'un problème en machine modifie quelque peu le problème initial; Il se peut que cette petite variation des données entraine une variation importante des résultats. C'est la notion de conditionnement d'un problème. On dit qu'un problème est bien (ou mal) conditionné, si une petite variation des données entraine une petite (une grande) variation sur les résultats.

Cette notion de conditionnement est liée au problème mathématique lui même et est indépendante de la méthode utilisée pour le résoudre. Une autre notion importante en pratique est celle de stabilité numérique.

Un problème peut être bien conditionné et la méthode utilisée pour le résoudre peut être sujette à une propagation importante des erreurs numériques. Ces notions de conditionnement d'un problème et de stabilité numérique d'une méthode de résolution sont fondamentales en analyse numérique. Si un problème est mal conditionné alors la solution exacte du problème tronqué ou arrondi à t digits pourra être très différente de la solution exacte du problème initial. Aucune methode ne pourra rien; il faudra essayer de donner une autre formulation au problème.

1.4 Les TP

1.4.1 TP 0: Rappels sur le langage MATLAB

1.4.2 TP 1: Systèmes de numérotation et gestion d'erreur

Exercice 1.4.1 ***Schéma de Horner***

Pour convertir un nombre $x = (x_p x_{p-1} ... x_1 x_0)_\beta$, *en base* 10, *il suffit d'évaluer dans cette base*

$$x = x_0 + x_1\beta + x_2\beta^2 + .. + x_p\beta^p.$$

Cette évaluation peut se faire en utilisant le schéma de Horner

$$x = \left(\left[... \left(x_p \beta + x_{p-1} \right) + ... \right] \beta + x1 \right) \beta + x0.$$

1. *Ecrire une fonction*

$$xDec = Horner\left(x, betha \right)$$

qui applique le schéma de Horner pour convertir en base 10 *l'entier dont la suite des chiffres en base* β *est donnée dans le tableau* x*. Tester cette fonction.*

2. *Effectuer les vérifications à l'aide des fonctions prédéfinies* ***bin2dec*** *et* ***base2dec.***

Exemple 1.4.1 Schéma de Horner

$(567)_8 = (5 \times 8 + 6) \times 8 + 7 = (375)_{10}$.

$(101101)_2 = ((((1 \times 2 + 0) \times 2 + 1) \times 2 + 1) \times 2 + 0) \times 2 + 1 = (45)_{10}$.

Solution 1.4.1 ***Schéma de Horner***

Selon le schéma de Horner, la valeur décimale $xDec$ *s'obtient en répétant pour les* n *chiffres donnés dans le tableau* x *l'opération* $xDec = xDec \times \beta + x(i)$*, pour* $i \in \{1, 2, ..., n\}$

Programme en Matlab:

```
function xDec=Horner(x,betha)
xDec=0 ;
for i=1 :1 :length(x)
xDec=xDec*betha+x(i) ;
end
```

Application:

```
>> Horner([1 0 1 1 0 1],2)
ans = 45
>> Horner([5 6 7],8)
ans = 375
```

Exercice 1.4.2 ***Convertir le nombre décimal*** x ***en base*** β

On obtient successivement la suite $(x_0, x_1, x_2, .., x_p)$ *des chiffres d'un entier* x *dans la base* β *de la manière suivante:*

- Le terme x_0 *est le reste de la division euclidienne de* x *par* β *car:* $x = x_0 + \beta\left(x_1 + x_2\beta + .. + x_p\beta^{p-1}\right)$

On note: $q_1 = x_1 + x_2\beta + .. + x_p\beta^{p-1}$, *le quotient de cette division euclidienne.*

- Le terme x_1 *est alors le reste de la division euclidienne de* q_1 *par* β.

- Plus généralement les termes x_i *pour* $i = 1, ..., p-1$, *sont les restes de la division euclidienne par* β *des quotients successifs.*

1. *Ecrire une fonction:*

$$y = cnvDec\_betha(x, betha),$$

qui convertit le nombre décimal x *en base* β. *Tester cette fonction.*

1. *Effectuer les vérifications à l'aide des fonctions prédéfinies* ***dec2bin*** *et* ***dec2base.***

Solution 1.4.2 ***Convertir le nombre décimal*** x ***en base*** β

Programme en Matlab:

```
function y=cnvDec_ betha(x,betha)
% La fonction cnvdec_ betha permmet de convertir l'entier décimal x
% en base bétha.
% A chaque itération, on calcule les quotients q(i+1) et le reste r(i)
% dans la division euclidienne de q(i) par bétha. On initialise q(1) à x.
q(1)=x; i=1;
while q(i)~=0
q(i+1)=fix(q(i)/betha); r(i)=q(i)-betha*q(i+1);
i=i+1;
end
% Le résultat s'obtient en inversant l'ordre des restes obtenus.
n=length(r); y=r(n:-1:1);
```

Fonctions prédéfinies de Matlab: *Les fonctions* ***dec2bin***, ***dec2hex*** *et* ***dec2base*** *permettent de convertir un entier donné en une chaîne de caractères représentant cet entier dans la base choisie. Les fonctions* ***bin2dec***, ***hex2dec*** *et* ***base2dec*** *effectuent la conversion inverse. Ainsi :*

>> dec2bin(55)

ans=110111

>> bin2dec('10111')

ans=23

>> base2dec('77',8)

ans=63

Les fonctions prédéfinies de conversion (dec2bin, bin2dec, etc...) ne s'appliquent qu'à des entiers.

Exercice 1.4.3 ***Conversion d'un nombre à virgule***

Ecrire une fonction:

$$s = fracDec\_Bin(frac, n);$$

qui calcule en base 2 *les* n *premiers chiffres de la partie fractionnaire* $frac$ *d'un réel donnée en base* 10 *. On obtient le résultat sous forme d'une chaîne de caractères* s*. Tester cette fonction.*

Solution 1.4.3 ***Conversion d'un nombre à virgule***

On utilise la variable Z *qui contiendra à chaque itération la partie fractionnaire* z_i *et la variable* X *qui contiendra la partie entière* x_i*. Suivant la valeur de* X*, on ajoute le caractère '0' ou '1' à la chaîne* s*.*

Programme en Matlab:

function s=fracDec_Bin(frac,n)

Z=frac ;

for i=1 :n,

*v=2*Z ; X=floor(v) ;*

Z=v-X ; %Z est la nouvelle partie fractionnaire

if X==1,s(i)=1;else s(i)=0;end

end

Exercice 1.4.4 ***Conversion d'un nombre à virgule***

Ecrire une fonction:

$$x = fracBin\_Dec(s);$$

qui convertit en valeur décimale x la chaîne de caractères s représentant la partie fractionnaire $(s_{-1}s_{-2}...s_{-n})$ d'un nombre en base 2. Tester cette fonction.

Solution 1.4.4 ***Conversion d'un nombre à virgule***

Pour $s = (s_{-1}s_{-2}...s_{-n})$, on doit calculer

$$s = s_{-1} \times 2^{-1} + s_{-2} \times 2^{-2} + ... + s_{-n} \times 2^{-n} = \sum_{i=1}^{i=n} s_{-i} \times 2^{-i}$$

On utilisera une variable P qui contient les valeurs successives de 2^{-i}. On initialise la valeur de x à 0 et, à chaque itération, on lui ajoute la valeur 2^{-i} seulement si $s(i) = 1$

Programme en Matlab:

function x=fracBin_Dec(s)

x=0;P =1/2 ;

for i = 1 :length(s)

if s(i)=='1' , x = x+P ; end

P = P/2 ;

end

Exercice 1.4.5 ***Erreur d'affectation***

Que se passe-t-il si on exécute avec Matlab les instructions suivantes ?

» disp('On ajoute 10 fois 0.1 ')

» S=0;

» for n=1:10, S=S+0.1;end

» if S==1

disp('et on trouve exactement 1')

else

disp('mais on ne trouve pas exactement 1')

end

Justiffier

Solution 1.4.5 ***Erreur d'affectation***

On obtient pour réponse :

On ajoute 10 fois 0.1

mais on ne trouve pas exactement 1

Lorsqu'on remplace le réel $0,1$ *par sa représentation machine, on commet une erreur due à la conversion en base 2 puis à l'arrondi*

Exercice 1.4.6 ***Erreur d'opérations***

1. *Estimer l'erreur absolue commise lorsqu'on effectue la somme:*

$$S = \underbrace{0,1 + 0,1 + ... + 0,1}_{1000\ termes}$$

2. *Vérifier en effectuant le calcul de cette somme avec Matlab , et en comparant avec le résultat exact.*

Solution 1.4.6 ***Erreur d'opérations***

1. *Le nombre décimal* $0,1$ *est converti et stocké en machine, avec une erreur relative majorée par* $\varepsilon_{machine} = 2^{-53}$*. L'erreur absolue sur la somme de* 1000 *nombres égaux à* $0,1$ *est donc estimée par*

$$1000 \times \varepsilon_{machine} \times (|0,1| + |0,1| + ... + |0,1|) = 1000^2 \times 2^{-53} \times 0,1$$

>> epsMachine=2^-53

epsMachine =1.1102e-016

*>> DeltaS=1000^2*epsMachine*0.1*

DeltaS = 1.1102e-011

2. *On effectue le calcul de S, puis de $|S - 100|$ avec Matlab:*

   >> S=0;

   >> for i=1:1000, S=S+0.1; end

   >> errObserv=abs(100-S)

   errObserv = 1.4069e-012

   L'erreur observée est bien inférieure à la majoration d'erreur DeltaS trouvée.

Exercice 1.4.7 ***Non associativité de l'addition machine***

On considére les trois nombres $x = 1$, $y = 2^{53}$, $z = -2^{53}$.

- *Donner les résultats des calculs pour :*

$s1 = (x + y) + z$ *et* $s2 = x + (y + z)$

- *Expliquer les résultats obtenus*

Solution 1.4.7 ***Non associativité de l'addition machine***

>> x=1; y=2^53; z=-2^53;

>> s1=(x+y)+z

s1 = 0

>> s2=x+(y+z)

s2 = 1

Dans le calcul de s_1 il y a eu absorption de x par y. Lorsqu'on ajoute z un phénomène de cancellation se produit. Dans le calcul de s_2, le phénomène de cancellation n'a pas lieu car on calcule $y + z$ avec les valeurs exactes de y et z.