*Give a digital computer a problem in arithmetic, and it will grind away methodically, tirelessly, at gigahertz speed, until ultimately it produces the wrong answer. (...) The problem is simply that computers are discrete and finite machines, and they cannot cope with some of the continuous and infinite aspects of mathematics.*
Brian Hayes, *A Lucid Interval*

CHAPITRE XV

# Calcul arrondi

**Objectif.** Dans ce chapitre notre but est de nous familiariser avec le calcul arrondi. Notamment nous voulons comprendre et illustrer ses avantages et ses pièges :

- Le calcul exact n'est pas toujours possible ; même quand il est possible, il n'est pas toujours efficace. Ainsi l'arrondi permet de rendre certains calculs faisables, ou bien plus efficaces sur ordinateur.
- En général le résultat d'un calcul arrondi sera erroné. Il est donc indispensable de connaître l'erreur commise, au moins de la majorer convenablement. La prudence s'impose !
- Sans aucun contrôle de la marge d'erreur la valeur numérique calculée n'apporte *aucune* information sur la valeur exacte cherchée. (J'insiste : *aucune*.)

**D'où vient le problème ?** Dans toute l'analyse mathématique le corps $\mathbb{R}$ des nombres réels joue un rôle primordial. Malheureusement l'implémentation de tels calculs sur ordinateur pose de sérieux problèmes. Tandis que les calculs avec les nombres entiers $\mathbb{Z}$ ou rationnels $\mathbb{Q}$ peuvent être effectués de manière exacte sur ordinateur, ceci est impossible pour les nombres réels. C'est cette difficulté qui distingue le calcul algébrique (exact) du calcul numérique (approché). La raison est simple :

***Limitation dans l'espace:*** Comme tout objet doit être représenté par une suite *finie* de bits 0 et 1, il est impossible de représenter tout nombre réel de manière exacte sur ordinateur.

***Limitation dans le temps:*** Les constructions en analyse utilisent la notion de limite « $u_n \to u$ pour $n \to \infty$ », alors que sur ordinateur on ne peut effectuer qu'un nombre *fini* d'opérations.

Si ces restrictions sont assez évidentes, les solutions possibles le sont beaucoup moins.

**Que peut-on faire ?** Dans les applications scientifiques ou technologiques on est souvent obligé de modéliser des calculs dans $\mathbb{R}$ sur ordinateur, malgré tout. Pour cela on utilise typiquement les *nombres à virgule flottante*, qui ne forment qu'un certain sous-ensemble fini $R \subset \mathbb{R}$ des réels. Ces nombres sont bien adaptés à l'ordinateur, mais se comportent assez différemment de ce que vous connaissez des mathématiques. Pour cette raison on les appelle aussi *nombres machine*. Le choix de $R$ n'est pas du tout naturel, il n'a aucune signification mathématique, et il ne suit que des considérations pragmatiques.

Peut-on néanmoins en déduire des informations sur le résultat réel cherché ? On verra qu'au moins dans des circonstances favorables la réponse est « oui », heureusement. Cet espoir explique l'usage fréquent du calcul numérique, mais il existe aussi de mauvais usages — qu'il faut éviter à tout prix.

**Pourquoi de petites erreurs sont-elles embêtantes ?** Parce qu'elles ne restent pas toujours petites ! Approcher les nombres réels par des nombres machine introduit des *erreurs d'arrondi*. C'est un problème inhérent et omniprésent du calcul numérique. Ces erreurs peuvent se propager dans les calculs, elles peuvent s'accumuler, voire exploser, ce qui peut rendre le résultat calculé inutilisable. C'est d'autant plus dangereux que l'utilisateur n'aura souvent aucune indication sur l'erreur commise et se fiera aveuglement à un résultat grossièrement faux ! Ceci arrive plus fréquemment que l'on ne pense. L'expérience montre qu'un utilisateur typique surestime systématiquement la précision des calculs numériques.

**Comment s'y prendre ?** Le bon sens et une certaine méfiance éclairée sont indispensables pour tout utilisateur, et ne serait-ce que pour interpréter avec prudence les résultats crachés par une application toute faite. Afin d'expliquer les quelques règles de bon sens, nous consacrons ce chapitre entier aux nombres à virgule flottante et leur calcul arrondi. Pour les raisons évoquées, le calcul arrondi est peu intuitif et l'usage des nombres flottants est assez délicat. Utilisés imprudemment, mêmes les calculs numériques les plus simples peuvent être grossièrement erronés. Pour vous en convaincre, ce chapitre vous présente de nombreux exemples, certes simplifiés mais assez réalistes. Il ne faut pas en conclure que le calcul

numérique soit inutile et se détourner avec mépris. Au contraire, il faut comprendre les fondements afin de calculer intelligemment avec ces nombres, puis interpréter correctement des résultats numériques.

**Que serait donc un usage réaliste ?** On rencontre souvent deux positions extrêmes :
– Le débutant imagine, à tort, que tout calcul numérique effectué sur ordinateur est correct, et qu'il peut s'y fier aveuglement. On va voir que ce point de vue naïf est *trop optimiste*.
– Après avoir vécu un certain nombre de mauvaises surprises, notre débutant résigne. Il pense maintenant, également à tort, que tout calcul numérique est grossièrement erroné, et qu'on ne peut jamais rien en conclure avec certitude. Ce point de vue est *trop pessimiste*.

Soyez assurés : le calcul numérique peut mener à des conclusions transparentes et prouvables. Mais comme tout outil informatique il nécessite un certain apprentissage. Notre objectif sera donc de développer un usage *réaliste*. Au prochain chapitre nous discuterons les éléments d'un *calcul fiable*, thème qui fait actuellement l'objet d'intenses recherches et qui sert déjà bien dans la pratique.

**Peut-on augmenter la précision ?** En C++ on dispose des types `float`, `double` et `long double` pour les nombres à virgule flottante. Leur précision est fixée à 24, 53 et 64 bits, respectivement, ce qui correspond à 8, 16 et 20 décimales environ. C'est suffisant pour beaucoup d'applications. Si jamais il vous faut plus de précision vous pouvez faire appel à une bibliothèque spécialisée, comme la *GNU multiple precision library* (GMP), qui implémente des nombres flottants en précision arbitraire (mais toujours finie).

Augmenter la précision n'est pourtant pas le remède à tous les maux numériques : d'une part cela alourdit les calculs, d'autre part il ne vous protège pas de possibles explosions d'erreurs. Le principe du calcul arrondi restera le même : bien que les erreurs d'arrondi soient plus petites, elles sont toujours non nulles, elles se propagent et parfois explosent, et toutes les difficultés mentionnées persisteront. On ne peut donc pas échapper aux problèmes fondamentaux de l'arithmétique arrondie.

**Pour en savoir plus.** On n'expliquera dans ce chapitre que l'idée essentielle des nombres flottants. Pour ne laisser rien à l'interprétation, le comportement des nombres machine a été standardisé en 1985, sous la norme « IEEE-754 : Standard for Binary Floating-Point Arithmetic ». Cet effort fut fondamental pour abolir l'anarchie qui régnait dans les divers implémentations et pour que le calcul numérique se comporte toujours de manière identique sur deux machines différentes. Littérature :

(1) Pour la petite histoire, on lira avec profit les mémoires de W. Kahan, un des initiateurs du standard IEEE-754 : `www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.
(2) Pour avoir une idée du standard actuel et futur (en cours de développement) vous pouvez consulter le site `grouper.ieee.org/groups/754/` et les liens y indiqués.
(3) Vous trouvez une excellente introduction à l'arithmétique flottante dans le cours de V. Lefèvre et P. Zimmermann, `www.vinc17.org/research/papers/arithflottante.pdf`.
(4) Vous pouvez également consulter les ouvrages de votre bibliothèque, par exemple le livre de J.-C. Bajard et J.-M. Muller, *Calcul et arithmétique des ordinateurs*, Lavoisier, Paris, 2004.

**Sommaire**

## 1. Motivation — quelques applications exemplaires du calcul numérique

Nous commençons par esquisser quelques applications assez représentatives du calcul numérique. Vous pouvez ainsi directement vous lancez dans la programmation, si cela vous intéresse et que vous en avez le loisir. Les chapitres suivants vous expliqueront quelques techniques de base pour analyser et améliorer de telles implémentations. Par conséquent, même si vous ne les programmez pas maintenant, gardez ces problèmes exemplaires en tête afin d'y revenir plus tard.

**1.1. Fonctions usuelles.** En mathématiques et dans de nombreuses applications vous utilisez fréquemment des fonctions « usuelles ». Si vous avez eu de la chance, quelques unes ont été construites et développées pour vous dans un cours d'analyse. Ensuite, pour les utiliser dans des calculs numériques, il faut encore les implémenter sur ordinateur. Comme toujours dans la programmation, le principal problème sera de le faire *correctement* et *efficacement*. On pourrait, bien sûr, se servir d'une bibliothèque de telles fonctions toute faite, mais essayons de voir si nous y arrivons nous-mêmes…

**Exemple 1.1** (repris au §3)**.** Écrire une fonction `sqrt(x)` qui calcule $\sqrt{x}$ à partir d'un nombre réel $x > 0$, en n'utilisant que les quatre opérations $+$, $-$, $*$, $/$. Documentez, vérifiez, puis testez soigneusement votre implémentation. Si vous voulez vous pouvez ensuite généraliser à une fonction `racine(x,n)` qui calcule $\sqrt[n]{x}$ pour $x > 0$ et $n \in \mathbb{N}$. Pouvez-vous majorer la marge d'erreur de vos résultats numériques ?

**Exemple 1.2** (repris au §5.5)**.** Écrire une fonction `exp(x)` qui calcule $e^x$ à partir d'un nombre réel $x \in \mathbb{R}$, en n'utilisant que les quatre opérations $+$, $-$, $*$, $/$. Documentez, vérifiez, puis testez soigneusement votre implémentation. Si vous voulez vous pouvez ensuite réfléchir sur `log(x)`, `sin(x)`, `cos(x)`, … ou vos fonctions préférées. Pouvez-vous majorer la marge d'erreur de vos résultats numériques ?

☞ *Ces notes ne peuvent être qu'une modeste invitation à la programmation numérique, en privilégiant des expériences pratiques. Elles partent du principe que vous disposez des bases requises en analyse et que vous les révisez en parallèle. On fera des rappels étape par étape, certes, mais en aucun cas ceux-ci ne peuvent remplacer un solide cours d'analyse.* ✍

**1.2. Résolution d'équations.** Les problèmes suivants sont classiques et seront discutés dans la suite.

**Exemple 1.3** (repris au §5.8)**.** Écrire un programme pour résoudre une équation quadratique $ax^2+bx+c = 0$. Le résultat calculé est-il raisonnablement proche de la solution exacte ? Comment le vérifier ? Votre programme sait-il traiter tous les cas ? Dans quels cas l'erreur devient-elle inacceptable ?

**Exemple 1.4** (repris au chapitre XVII)**.** Écrire un programme pour résoudre une équation polynomiale de la forme $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = 0$. Alors qu'en degré $n \leq 4$ il existe des « formules closes », ceci n'est plus le cas en degré $n \geq 5$. Y-a-t il toujours des solutions ? Combien ? Comment les trouver ? Une fois trouvées, comment vérifier la qualité des solutions approchées ?

**Exemple 1.5** (repris au chapitre XVIII)**.** Comment résoudre une équation linéaire $Ax = b$ où $A$ est une matrice, $b$ est un vecteur donné, et le vecteur $x$ est inconnu ? Y a-t-il toujours une solution ? Est-elle unique ? Si oui, comment la trouver ? Une fois trouvée, comment vérifier la qualité d'une solution approchée ?

**1.3. Intégration numérique.** Assez souvent, quand le calcul exact est impossible ou trop dur, on veut numériquement calculer une intégrale de la forme $I = \int_a^b f(x)dx$, par exemple $\int_0^1 e^{-x^2/2}dx$. Dans ce cas, au lieu de l'intégrale exacte, on pourra regarder l'approximation par la $n$ième somme de Riemann

$$I_n := I_n(f,a,b) = \sum_{k=1}^{n} f(x_k)\Delta x \qquad \text{avec} \quad \Delta x = \frac{b-a}{n} \quad \text{et} \quad x_k = a + k\Delta x$$

**Théorème 1.6.** *Si $f\colon [a,b] \to \mathbb{R}$ est continue, alors $I_n \to I = \int_a^b f(x)dx$ pour $n \to \infty$.*

**Exercice/M 1.7.** Faire un dessin pour motiver l'approximation de l'intégrale $I$ par la somme $I_n$. En passant vous êtes invités à réviser le résultat précédent dans votre cours d'analyse. Dans notre exemple, $f$ est monotone ; donner un encadrement explicite de $I$ en fonction de $I_n$ qui prouve que $I_n \to I$.

**Exemple 1.8.** Écrire une fonction `riemann(a,b,n)` qui calcule $I_n$ pour une fonction $f\colon [a,b]\to\mathbb{R}$ fixée. Documentez, vérifiez, puis testez soigneusement votre implémentation. Pouvez-vous majorer la marge d'erreur de vos résultats numériques ? Qu'obtenez vous dans notre exemple $\int_0^1 e^{-x^2/2}dx$ ? Quand vous faites augmenter $n=2,4,8,16,32,\dots$ est-ce que les approximations successives se comportent comme prévues ? Quels problèmes constatez-vous ? Mettent-ils en question le théorème ? Ou plutôt nos méthodes de calculs ? Essayez de décrire les problèmes aussi précisément que possible, puis esquisser quelques solutions ou pistes qui vous semblent prometteuses.

**1.4. Systèmes dynamiques.** Comme vous savez peut-être, le mouvement de deux corps sous la force gravitationelle peut être décrit par une jolie formule close. Pour un nombre $n\geq 3$ de corps, par contre, un tel traitement est en général impossible et il faut faire appel à des méthodes numériques. Si cela vous intéresse, nous esquissons ici les ingrédients d'une telle implémentation assez simple.

Beaucoup de systèmes en physique, chimie, biologie, climatologie, etc. sont modélisés d'une manière très similaire. Historiquement, le mouvement planétaire fut un des premiers exemples à être étudié, et reste à ce jour un sujet classique que tout étudiant en sciences devrait connaître. Bien sûr vous pouvez le remplacer par un autre qui vous est plus proche.

*Un modèle de la mécanique céleste.* — On considère $n$ corps, numérotés $i=1,\dots,n$, chacun d'une certaine masse constante $m_i>0$. On s'intéresse à la position $x_i=x_i(t)\in\mathbb{R}^3$ et la vitesse $v_i=v_i(t)\in\mathbb{R}^3$ au cours du temps $t\geq 0$. Au temps $t=0$ les positions initiales $x_i(0)=x_i^0$ et les vitesses $v_i(0)=v_i^0$ sont données, puis elles évoluent suivant la mécanique newtonienne : on a $x_i'(t)=v_i(t)$, puis la vitesse $v_i(t)$ change sous l'influence gravitationelle des autres corps. Plus précisément, la force gravitationelle sur le corps $i$ est donnée par $F_i=\sum_{j\neq i}\gamma m_i m_j\frac{(x_j-x_i)}{|x_j-x_i|^3}$, avec une certaine constante universelle $\gamma$, et on a $m_i v_i'=F_i$.

*Le système à résoudre.* — Calculer les trajectoires de $n$ planètes, c'est donc résoudre le système d'équations différentielles $x_i'=v_i$ et $v_i'=F_i/m_i$ avec les conditions initiales $x_i(0)=x_i^0$ et $v_i(0)=v_i^0$. Évidemment on va supposer $x_i\neq x_j$ pour $i\neq j$ et tout $t\geq 0$, sinon notre modèle va tout droit dans la catastrophe. Si jamais un tel problème se produira lors des calculs, on abandonnera en expliquant brièvement ce qui s'est passé.

*Discrétisation du temps.* — Le modèle continu ne peut pas s'implémenter directement sur ordinateur. Nous allons donc discrétiser le temps $t$ en intervalles d'une longueur fixée $\Delta t$. Autrement dit, au lieu d'un temps continue $t\in\mathbb{R}_+$ nous regardons un temps discret $t_k=k\Delta t$ pour $k=0,1,2,3,\dots$.

*Approximation numérique.* — Une fois discrétisé, le système ci-dessus se transforme en une simple formule de récurrence : nous avons $x_i'\approx\frac{x_i(t_{k+1})-x_i(t_k)}{\Delta t}$, ce qui nous mène à $x_i(t_{k+1})\approx x_i(t_k)+v_i(t_k)\Delta t$. De même $v_i'\approx\frac{\Delta v_i}{\Delta t}$ nous mène à $v_i(t_{k+1})\approx v_i(t_k)+\Delta t F_i(t_k)/m_i$. Nous obtenons ainsi à nouveau l'approximation d'une intégrale :

$$x_i(t_{k+1})=x_i(t_k)+v_i(t_k)\Delta t\quad\text{et}\quad v_i(t_{k+1})=v_i(t_k)+\sum_{j\neq i}\gamma m_j\frac{x_j-x_i}{|x_j-x_i|^3}\Delta t.$$

*L'espoir tacite de stabilité.* — Bien entendu notre reformulation numérique s'est éloignée du système exact, mais nous pouvons espérer que les trajectoires calculées et les trajectoires exactes se ressemblent. Pour le moment nul ne nous garantit que cet espoir soit bien fondé. Les résultats numériques seront donc à vérifier et à interpréter avec la plus grande prudence !

**Exemple 1.9.** Si cela vous intéresse vous pouvez implémenter l'approximation numérique esquissée ci-dessus. (Pour faire joli il serait souhaitable d'ajouter un affichage graphique…) Documentez, vérifiez, puis testez soigneusement votre implémentation. Expérimenter avec le choix du paramètre $\Delta t$. Est-ce que la trajectoire calculée en dépend ? Quels sont les avantages et les inconvénients de choisir $\Delta t$ plus petit ou plus grand ? Voyez-vous des situations où le modèle numérique se comporte raisonnablement ? Pouvez-vous concevoir des situations où le calcul numérique échoue ? Quelles en sont les causes possibles ?

☞ *Invariants :* Pour vérification automatique par votre logiciel, vous pouvez faire calculer l'énergie et la quantité du mouvement totales. Dans le modèle exacte ces quantités restent constantes au cours du temps, et une bonne approximation doit faire pareil. Il peut y avoir d'autres invariants. À noter aussi que ce n'est qu'une exigence minimale, et non une garantie de correction.

## 2. Le problème de la stabilité numérique

**2.1. Un exemple simple : les suites de Fibonacci.** Comme premier exemple, simple et concret, nous allons regarder des suites de Fibonacci : on se donne deux valeurs initiales $x_0, x_1 \in \mathbb{R}$ puis on procède par la récurrence $x_{n+2} = x_{n+1} + x_n$. Pour un système dynamique on ne peut plus simple !

**Exemple 2.1.** Pour $x_0 = 1$ et $x_1 = 1$ on obtient la célèbre suite de Fibonacci $x_2 = 2$, $x_3 = 3$, $x_4 = 5$, $x_5 = 8$, etc. Regardons ce qui se passe si l'on varie légèrement les conditions initiales, disons $x'_1 = 1.01$, ce qui est une déviation de 1%. Suivant la récurrence on trouve :

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_n$ | 1.00 | 1.00 | 2.00 | 3.00 | 5.00 | 8.00 | 13.00 | 21.00 | 34.00 | 55.00 | 89.00 |
| $x'_n$ | 1.00 | 1.01 | 2.01 | 3.02 | 5.03 | 8.05 | 13.08 | 21.13 | 34.21 | 55.34 | 89.55 |

On constate qu'ici l'erreur initiale se propage d'une manière bien contrôlée : dans toute la suite les valeurs $x_n$ et $x'_n$ ne diffèrent que de 1%. Vous pouvez tester cette observation sur d'autres exemples à l'aide du programme `fibonacci.cc`.

**Définition 2.2** (stabilité, formulation heuristique)**.** On dit qu'un calcul numérique est *stable* lors qu'un petit changement des données initiales n'entraîne qu'un petit changement des résultats.

**Exemple 2.3.** Malheureusement pas tous les calculs se réjouissent de cette bonne propriété de stabilité. Même la récurrence de Fibonacci peut être numériquement méchante. Par exemple, on constate un phénomène étrange pour $x_0 = 1$ et $x_1 \approx -0.618$ :

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_n$ | 1.000 | −0.618 | 0.382 | −0.236 | 0.146 | −0.090 | 0.056 | −0.034 | 0.022 | −0.012 | 0.010 |
| $x'_n$ | 1.000 | −0.619 | 0.381 | −0.238 | 0.143 | −0.095 | 0.048 | −0.047 | 0.001 | −0.046 | −0.045 |

Dans la suite on voit que l'erreur croît de plus en plus rapidement : on trouve $x_{20} = 0.230$ et $x'_{20} = -6.535$, puis $x_{30} = 28.280$ et $x'_{30} = -803.760$. Ceci veut dire qu'une petite erreur initiale (moins de 1%) peut se propager et s'amplifier, et finalement entraîner une erreur considérable au cours de quelques itérations.

**Remarque 2.4.** Souvent les valeurs avec lesquelles on calcule ne sont pas exactes :

- Si les valeurs initiales sont issues d'un calcul arrondi, elles sont en général déjà contaminées d'erreurs d'arrondi. Le mieux que l'on puisse espérer est que le calcul précédent permet explicitement de garantir une certaine précision.
- Si les données initiales sont des quantités réelles mesurées (par une expérience physique ou autre), elles ne peuvent non plus être exactes. Dans ce cas il est nécessaire de spécifier la précision, disons sous forme d'un « intervalle de confiance ».
- Même dans les rares cas où les données initiales sont exactes, les calculs suivants introduiront des erreurs d'arrondi. On n'échappera donc pas à des perturbations des données, qui font que les calculs peuvent s'éloigner des résultats exacts.

Dans une telle situation *numériquement instable* un calcul ne sert à rien : la moindre erreur peut exploser au cours du calcul, de sorte que le résultat calculé n'apporte plus aucune information.

☞ *Des calculs numériques sans contrôle d'erreur sont toujours périlleux. Dans une situation instable ils tournent à la catastrophe car la moindre perturbation peut s'amplifier au cours du calcul. Des résultats ainsi calculés sont aussi fiables qu'un tirage au sort.* ✍

**2.2. Analyse mathématique du phénomène.** À titre d'illustration, analysons une question naturelle :

**Exemple 2.5.** Que peut-on dire de la convergence ou divergence d'une suite de Fibonacci en fonction des valeurs initiales $x_0$ et $x_1$ ? En fixant $x_0 = 1$, est-il possible de choisir $x_1$ de sorte que l'on obtienne une suite convergente ? Si oui, quelle sera la limite ?

- Pour $x_0 = 1$ et $x_1 = 1$ on trouve $x_n \to +\infty$. (Pourquoi ?)
- Pour $x_0 = 1$ et $x_1 = -1$ on trouve $x_n \to -\infty$. (Le vérifier.)

Vue les valeurs numériques ci-dessus, on peut soupçonner que pour $x_1 \geq -0.618$ la suite s'échappe vers $+\infty$, alors que pour $x_1 \leq -0.619$ elle s'échappe vers $-\infty$. Peut-être trouve-t-on une valeur intermédiaire qui donne lieu à une suite convergente ?

Certes, l'approche par tâtonnement devient vite fastidieuse. Heureusement il est possible de répondre à cette question dès que nous disposons d'une formule close pour le terme général $x_n$ :

**Proposition 2.6.** *Toute suite vérifiant $x_{n+2} = x_{n+1} + x_n$ pour tout $n \geq 0$ est de la forme $x_n = a\alpha^n + b\beta^n$ avec des coefficients $a, b \in \mathbb{R}$ et des constantes $\alpha = \frac{1+\sqrt{5}}{2}$ et $\beta = \frac{1-\sqrt{5}}{2}$. En voici deux conséquences :*

(1) *Pour les deux premiers termes on obtient $x_0 = a + b$ et $x_1 = a\alpha + b\beta$, ce qui permet de trouver $a, b$ en fonction de $x_0, x_1$. Concrètement, en fixant $x_0 = 1$, on obtient $a = \frac{x_1-\beta}{\alpha-\beta}$ et $b = 1 - a$.*

(2) *Pour $a \neq 0$ la suite $(x_n)_{n\in\mathbb{N}}$ diverge, plus précisément on voit que $x_n \to +\infty$ pour $a > 0$, et $x_n \to -\infty$ pour $a < 0$. Le cas critique $a = 0$ se produit si et seulement si $x_1 = \beta x_0$ : dans ce cas on a convergence vers zéro, $x_n = b\beta^n \to 0$ pour $n \to \infty$.*

**Exercice/M 2.7.** Montrer la proposition précédente.

**Exercice 2.8.** Le programme `fibonacci.cc` permet de calculer une suite de Fibonacci à partir de $x_0$ et $x_1$. Essayez de comprendre son fonctionnement, puis expérimentez avec ce programme.

(1) Quand vous vérifiez les calculs pour les tableaux ci-dessus, vous allez constater de petites différences étranges : alors que les valeurs initiales sont connues de manière exacte, le calcul des termes suivants est contaminé d'erreurs d'arrondi. A priori on ne s'y attend pas dans un calcul aussi simple, avec des « chiffres ronds ». Ceci s'expliquera plus loin : le programme repose sur les nombres à virgule flottante, qui ont un comportement particulier et qui font l'objet de ce chapitre.

(2) Qu'observez-vous autour du cas critique, $x_0 = 1$ et $x_1 \approx \beta \approx -0.6180339887498949$ ? Les premières valeurs calculées $x_2, x_3, x_4, \ldots$ semblent-elles plausibles, autant que l'on puisse dire des décimales affichées ? Les valeurs absolues $|x_n|$ décroissent-elles, comme prévu ? À partir de $x_{50}$ (voire $x_{100}$) le calcul numérique déraille ; comment expliquer cette explosion d'erreurs ?

(3) Est-il possible de construire sur ordinateur une suite de Fibonacci convergente ? Quelle est la difficulté ? Dans quel sens le calcul numérique (utilisant les nombres machine) peut-il correspondre au raisonnement mathématique (concernant les nombres réels) ? Dans quelle mesure peut-on faire confiance à notre calcul numérique ?

**2.3. Conclusion.** Ce joli exemple est trop simpliste dans le sens qu'il est linéaire et vous disposez d'une formuler close. Ainsi vous pouvez analysez la situation dans le moindre détail, et en particulier la propagation des erreurs se comprend parfaitement. Dans un exemple réaliste la situation sera plus compliquée, mais qualitativement les mêmes phénomènes peuvent se produire, quoique de façon moins transparente. Ainsi pour tout calcul numérique qui se respecte il sera question de stabilité et de propagation d'erreurs.

## 3. Le problème de l'efficacité : calcul exact vs calcul arrondi

En analyse en construit certaines fonctions $f : \mathbb{R} \supset U \to \mathbb{R}$, comme $\sqrt[n]{x}$, $\sin(x)$, $\cos(x)$, $\exp(x)$, $\log(x)$, etc. Pour le calcul numérique sur ordinateur, nous aimerions calculer explicitement la valeur $f(x)$ pour un $x$ donné. Typiquement cela ne sera pas possible de manière exacte, mais on peut espérer d'implémenter un calcul approché, si possible efficace.

**3.1. La méthode de Newton-Héron.** Pour entrer dans le vif du sujet, nous allons illustrer notre propos par une méthode importante et peu triviale : le calcul approché d'une racine $\sqrt[n]{a}$ d'un nombre réel $a > 0$. La méthode de Newton nous donne un formidable outil pour de telles approximations, car elle fournit une suite $(u_k)_{k\in\mathbb{N}}$ facilement calculable qui converge rapidement vers la racine cherchée.

**Proposition 3.1** (Newton-Héron, version qualitative)**.** *Soient $n \geq 2$ un entier et $a \in \mathbb{R}_+$ un nombre réel positif. Pour toute valeur initiale $u_0 > 0$ la suite récurrente $u_{k+1} = \frac{1}{n}\left((n-1)u_k + a/u_k^{n-1}\right)$ converge vers la racine $r = \sqrt[n]{a}$, c'est-à-dire vers l'unique nombre réel $r \in \mathbb{R}_+$ vérifiant $r^n = a$.*

Vous connaissez ce résultat sans doute de votre cours d'analyse. À noter qu'il s'agit d'une affirmation topologique, à caractère purement qualitatif. Pour le calcul numérique il est indispensable de l'affiner par une majoration d'erreur (résultat métrique, à caractère quantitatif). Un peu mieux encore, l'énoncé suivant donne un encadrement (ce qui repose sur une structure encore plus fine : la relation d'ordre sur $\mathbb{R}$).

**Théorème 3.2** (Newton-Héron, version quantitative)**.** *Soient $n \geq 2$ un entier et $a \in \mathbb{R}_+$ un nombre réel positif. Pour toute valeur initiale $u_0 > 0$ la suite récurrente $u_{k+1} = \frac{1}{n}\big((n-1)u_k + a/u_k^{n-1}\big)$ converge vers la racine $r = \sqrt[n]{a}$. Pour $k \geq 1$ on a convergence monotone $u_k \searrow r$. Symétriquement pour $v_k = a/u_k^{n-1}$ on a $v_k \nearrow r$. On obtient ainsi des encadrements explicites $v_k \leq r \leq u_k$ de plus en plus fins :*

$$v_1 \leq v_2 \leq v_3 \leq \cdots \leq r \leq \cdots \leq u_3 \leq u_2 \leq u_1$$

*Quant à la vitesse de convergence, l'écart relatif $\varepsilon_k = \frac{u_k - r}{r}$ vérifie*

$$\varepsilon_{k+1} \leq \min\big(\, \tfrac{n-1}{2}\varepsilon_k^2 \,,\, \tfrac{n-1}{n}\varepsilon_k \,\big)$$

*Pour une valeur $u_k$ loin de la racine $r$, la convergence est donc au moins linéaire, avec un rapport de contraction $\frac{n-1}{n} < 1$. Finalement, pour $u_k$ proche de la racine $r$ (à savoir pour $r \leq u_k \leq \frac{n+2}{n}r$) la convergence est quadratique : à chaque itération le nombre de décimales valables double à peu près.*

**Remarque 3.3.** Pas tous les problèmes numériques admettent de solutions aussi élégantes et efficaces. Pour la mettre en relief, soulignons donc trois points forts de la méthode de Newton-Héron :

- C'est la convergence quadratique qui fait de ce théorème un outil très puissant : si vous avez calculé un encadrement $[v_k, u_k]$ à $\approx 10^{-2}$ près, disons, l'itération suivante ne laissera qu'un écart $\approx 10^{-4}$, celle d'après $\approx 10^{-8}$, puis $\approx 10^{-16}$ etc.
- Non seulement le théorème précédent vous garantit une convergence rapide, mais vous pouvez à tout moment, par un simple calcul, contrôler l'écart restant $|u_k - v_k|$. Ceci vous permet de terminer l'itération lorsque la précision est suffisante pour vos besoins.
- De plus la méthode est numériquement stable : si la valeur approchée $u_k$ est perturbée par une erreur d'arrondi, les itérations suivantes convergeront tout de même.

**Exercice/M 3.4.** La dynamique de l'itération de Newton est une jolie application de l'étude de fonctions. Comme exercice, vous pouvez prouver le théorème en détaillant l'esquisse suivante.

ESQUISSE DE PREUVE. La fonction $p\colon \mathbb{R}_+ \to \mathbb{R}_+$, $x \mapsto x^n$ est strictement croissante, donc injective. Elle est continue avec $p(0) = 0$ et $p(x) \to \infty$ pour $x \to \infty$, donc surjective par le théorème des valeurs intermédiaires. Autrement dit, il s'agit d'une bijection de $\mathbb{R}_+$ sur $\mathbb{R}_+$ : pour tout $a \in \mathbb{R}_+$ il existe un et un seul réel positif $r \in \mathbb{R}_+$ vérifiant $r^n = a$. Pour approcher la valeur $r$ cherchée, nous itérons la fonction

$$f\colon \mathbb{R}_+ \to \mathbb{R}_+ \quad \text{donnée par} \quad f(x) = \frac{1}{n}\big((n-1)x + a/x^{n-1}\big).$$

L'unique point fixe de $f$ est $r$. Elle est dérivable de classe $C^\infty$, avec $f'(x) = \frac{n-1}{n}(1 - a/x^n)$. Sur $]0, r[$ on a $f' < 0$ et la fonction $f$ est strictement décroissante et vérifie donc $f(x) > f(r) = r$. Sur $]r, +\infty[$ on a $0 < f'(x) < \frac{n-1}{n}$ et la fonction $f$ est strictement croissante. Par le théorème des accroissements finis on a $f(x) - r = f(x) - f(r) = f'(\xi)(x - r) \leq \frac{n-1}{n}(x - r)$ donc $r < f(x) < x$. On conclut que le point fixe $r$ est attractif, avec tout $\mathbb{R}_+$ pour bassin d'attraction. Les suites $v_k$ et $u_k$ donnent ainsi des encadrements

$$v_1 \leq v_2 \leq v_3 \leq \cdots \leq r \leq \cdots \leq u_3 \leq u_2 \leq u_1.$$

Par conséquent, les limites $v = \lim v_k$ et $u = \lim u_k$ existent et vérifient $v \leq r \leq u$. D'autre part, la continuité de $f$ et l'équation de récurrence $u_{k+1} = f(u_k)$ donnent, par passage à la limite,

$$u = \lim u_{k+1} = \lim f(u_k) = f(\lim u_k) = f(u).$$

On conclut que $u = r$ par unicité du point fixe. De manière analogue $v = r$.

Étudions finalement la vitesse de convergence. Pour tout $k \geq 1$ nous avons $u_k = r(1 + \varepsilon_k)$ avec une erreur relative $\varepsilon_k \geq 0$. Après un petit calcul on trouve que $\varepsilon_{k+1} = g(\varepsilon_k)$ est obtenue en itérant la fonction

$$g(\varepsilon) = \tfrac{n-1}{n}\varepsilon + \tfrac{1}{n}\big[(1+\varepsilon)^{1-n} - 1\big].$$

Évidemment $g$ est majorée par $\frac{n-1}{n}\varepsilon$, ce qui donne la convergence linéaire. Mais $g$ est aussi majorée par $h(\varepsilon) = \frac{n-1}{2}\varepsilon^2$, ce qui assure la convergence quadratique. Effectivement, on a $g(0) = h(0) = 0$ et $g'(0) = h'(0) = 0$ ainsi que $g''(\varepsilon) \leq h''(\varepsilon)$ pour tout $\varepsilon \geq 0$, ce qui implique $g' \leq h'$ puis $g \leq h$. □

**3.2. Exemples numériques.**

**Exemple 3.5.** Calculons par exemple des valeurs approchées de $r = \sqrt{2}$ à $10^{-10}$ près. Avec la valeur initiale $u_0 = 1$ l'itération de Newton donne les encadrements suivants :

$$\begin{aligned} \tfrac{4}{3} \le r \le \tfrac{3}{2} \qquad & 1.3333333333 \le r \le 1.5000000000 \\ \tfrac{24}{17} \le r \le \tfrac{17}{12} \qquad & 1.4117647058 \le r \le 1.4166666667 \\ \tfrac{816}{577} \le r \le \tfrac{577}{408} \qquad & 1.4142114384 \le r \le 1.4142156863 \\ \tfrac{941664}{665857} \le r \le \tfrac{665857}{470832} \qquad & 1.4142135623 \le r \le 1.4142135624 \end{aligned}$$

Dans la dernière ligne on a $v_4 - u_4 \le 10^{-10}$. À noter que les valeurs dans $\mathbb{Q}$ à gauche sont exactes, alors que les développements décimaux à droite sont arrondis à 10 décimales : vers le bas pour $v_k$ et vers le haut pour $u_k$, afin de garantir la correction de l'encadrement affiché.

**Exemple 3.6.** Dans l'exemple précédent, le calcul dans $\mathbb{Q}$ semble satisfaisant. En général il n'en est rien ! Regardons le calcul de $r = \sqrt[3]{10}$ à $10^{-10}$ près. Avec la valeur initiale $u_0 = 1$ l'itération de Newton donne :

$$\begin{aligned} \tfrac{5}{8} &\le r \le \tfrac{4}{1} \\ \tfrac{640}{529} &\le r \le \tfrac{23}{8} \\ \tfrac{44774560}{24098281} &\le r \le \tfrac{4909}{2116} \\ \tfrac{6500450623479657648040}{3049614553054553981809} &\le r \le \tfrac{55223315303}{25495981298} \\ \tfrac{151137897149456207062734071576976875584120695784505967636052968 10}{701559854671230732015021361552743850496833886768021475309468 6841} &\le r \le \tfrac{837591699261179839454692621670 29}{388764578053937685469668481040 41} \end{aligned}$$

Après cinq itérations la précision $v_5 - u_5 \approx 0,0001837$ est encore loin d'être satisfaisante, mais les fractions produites dans le calcul commencent déjà à exploser. Il faut encore deux itérations pour atteindre la précision souhaitée, et les numérateurs et dénominateurs ont quelques centaines de chiffres. (Inutile de les reproduire ici, mais vous êtes invités à le vérifier sur ordinateur.)

**Exemple 3.7.** On peut calculer directement avec des développements décimaux sous forme de nombres à virgule flottante, en utilisant l'arithmétique arrondie expliquée plus bas. Nous obtenons ainsi les encadrements suivants de la racine cherchée $r = \sqrt[3]{10}$ :

$$\begin{aligned} 0.6250000000000000 \le r \le 4.0000000000000000 \\ 1.2098298676748582 \le r \le 2.8750000000000000 \\ 1.8579980870834728 \le r \le 2.3199432892249528 \\ 2.1315646651045386 \le r \le 2.1659615551777928 \\ 2.1543122250101293 \le r \le 2.1544959251533748 \\ 2.1544346865510652 \le r \le 2.1544346917722930 \\ 2.1544346900318837 \le r \le 2.1544346900318838 \end{aligned}$$

Après sept itérations nous arrivons ainsi à une précision $< 10^{-16}$, comme souhaité.

**3.3. Conclusion.** Soulignons à nouveau que les nombres rationnels $\mathbb{Q}$ forment un corps, ce qui veut dire que vous pouvez effectuer les quatre opérations de base $+, -, \cdot, /$ comme vous les connaissez. Ajoutons que l'on peut représenter tout nombre rationnel de manière exacte sur ordinateur, typiquement sous forme de numérateur et dénominateur, et ainsi les calculs dans $\mathbb{Q}$ s'effectuent de manière exacte.

En analyse on s'intéresse surtout au corps $\mathbb{R}$ des nombres réels. Ici une représentation exacte sur ordinateur est en général impossible, mais on peut approcher n'importe quel nombre réel par des rationnels. Cette propriété est fondamentale pour la théorie et aussi pour des calculs numériques.

Jusqu'ici tout marche bien, mais malheureusement les nombres rationnels entraînent assez souvent des calculs inutilement lourds. La catastrophe de l'exemple 3.6 illustre que le calcul exact dans $\mathbb{Q}$, bien que théoriquement préférable, peut être mal adapté et inefficace pour certaines tâches. Ce phénomène est assez fréquent dans les calculs itératifs : lors d'un calcul dans $\mathbb{Q}$ les fractions peuvent exploser avant que l'on n'arrive à un résultat satisfaisant (suffisamment proche de la limite cherchée). Dans ce cas la manipulation des nombres rationnels devient trop coûteuse en temps et mémoire.

## 4. Qu'est-ce que les nombres à virgule flottante ?

Par souci d'efficacité il semble avantageux de calculer avec un développement décimal convenablement arrondi à chaque étape. De toute manière, c'est souvent ce format qui est souhaité pour le résultat final. Dans ces circonstances on abandonne le calcul exact au profit des *nombres à virgule flottante*. C'est une astucieuse invention de l'informatique qui permet des calculs efficaces. Hélas, le prix à payer sont les erreurs d'arrondi. Par conséquent il faut comprendre leur fonctionnement et quelques règles de bon sens afin d'utiliser intelligemment cet outil informatique et d'interpréter correctement ses résultats.

**4.1. Développement binaire.** Regardons le développement binaire d'un nombre réel $x \in [1,2]$ :

$$x = \langle 1.a_1a_2a_3a_4\ldots\rangle_{\text{bin}} := 1 + \sum_{k=1}^{\infty} a_k 2^{-k} \quad \text{avec des chiffres binaires } a_k \in \{0,1\}. \tag{1}$$

Toute telle série converge vers un nombre réel $x \in [1,2]$, et réciproquement tout $x \in [1,2]$ peut être représenté par un tel développement. (Certains nombres en admettent deux — lesquels ?) Pour recouvrir tout $\mathbb{R}_+$ on multiplie par un facteur $2^e$ avec exposant $e \in \mathbb{Z}$, puis on ajoute un signe pour atteindre $\mathbb{R}_-$ :

$$x = \pm\langle 1.a_1a_2a_3a_4\ldots\rangle_{\text{bin}} \cdot 2^e. \tag{2}$$

La représentation (1) est *à virgule fixe*, alors que la représentation (2) est *à virgule flottante*, parce que le facteur $2^e$ permet de décaler la virgule, c'est-à-dire de la rendre « flottante ». En base 10 par exemple on a $1234,567 = 123,4567 \cdot 10^1 = 12,34567 \cdot 10^2 = 1,234567 \cdot 10^3$. On peut ainsi supposer que la virgule se trouve immédiatement après le premier chiffre non nul, ce qui rend cette représentation unique.

**4.2. Nombres à virgule flottante.** Sur ordinateur on ne peut pas stocker une suite *infinie* de chiffres. On fixe donc une longueur $\ell$ et on ne considère que les nombres réels qui s'écrivent

$$\pm\,\langle 1.a_1a_2a_3\ldots a_\ell\rangle_{\text{bin}} \cdot 2^e. \tag{3}$$

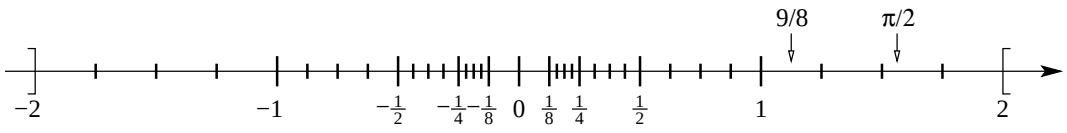
La suite des chiffres $m = (1, a_1, a_2, a_3, \ldots, a_\ell)$ est appelée la *mantisse*, et $\ell+1$ est donc la *longueur* de la mantisse. Très souvent on restreint aussi l'exposant $e$ à un intervalle $[\![e_{\min}, e_{\max}]\!]$ fixé d'avance. La définition des nombres à virgule flottante dépend donc du choix des trois paramètres $\ell, e_{\min}, e_{\max}$.

**Définition 4.1.** L'ensemble $R$ formé de 0 et des nombres de la forme (3) est *l'ensemble des nombres exactement représentables* avec une mantisse de longueur $\ell+1$ et un exposant $e \in [\![e_{\min}, e_{\max}]\!]$.

$$R = \{0\} \cup \left\{ \pm\left(1 + \frac{m}{2^\ell}\right) \cdot 2^e \mid m \in [\![0, 2^\ell[\![,\ e \in [\![e_{\min}, e_{\max}]\!] \right\}$$

On les appelle aussi *nombres à virgule flottante*, ou un peu plus court *nombres flottants*, ou encore *nombres machine*. Il s'agit d'un sous-ensemble fini de $\mathbb{R}$. À noter que $R$ ne forme pas un sous-corps de $\mathbb{R}$ et que $R$ n'a aucune propriété mathématique intéressante — mise à part la simplicité de ses éléments en développement binaire.

**Exemple 4.2.** Avec une mantisse de longueur $\ell+1 = 3$ et un exposant $e \in [\![-3, 0]\!]$, les nombres exactement représentables sont $R = \{0\} \cup \{\pm 1\} \cdot \left\{\frac{4}{4}, \frac{5}{4}, \frac{6}{4}, \frac{7}{4}\right\} \cdot \left\{2^{-3}, 2^{-2}, 2^{-1}, 2^0\right\}$. Graphiquement, ceci donne la discrétisation suivante de l'intervalle $]-2,2[$, qui ressemble vaguement à une *échelle logarithmique* :




On voit par exemple que $x = 9/8$ ne peut être représenté par un tel nombre machine : il faut l'approcher par un de ses voisins $\lfloor x \rfloor_R = 1,0$ ou $\lceil x \rceil_R = 1,25$. Il en est de même pour le nombre $\pi/2$ qui peut être approché par son voisin le plus proche $\lfloor \pi/2 \rceil_R = 1,5$, par exemple. Ainsi le choix des nombres machine introduit des erreurs d'arrondi inévitables. Typiquement il faut s'attendre à une erreur relative $\frac{|\hat{x}-x|}{|x|} \approx 2^{-\ell}$.

**4.3. Les types primitifs du C++.** Augmenter la longueur $\ell$ de la mantisse rend la discrétisation plus fine. Élargir l'intervalle de l'exposant $[\![e_{\min}, e_{\max}]\!]$ étend l'intervalle de la discrétisation. (Le détailler sur des exemples.) En C++ on dispose des types `float`, `double` et `long double` pour les nombres à virgule flottante, correspondant aux paramètres suivants :

| type | mantisse | erreur relative | exposant | minimum | maximum |
|---|---|---|---|---|---|
| `float` | 24 bits | $2^{-24} \approx 6 \cdot 10^{-8}$ | 8 bits | $2^{-128} \approx 10^{-38}$ | $2^{127} \approx 10^{38}$ |
| `double` | 53 bits | $2^{-53} \approx 1 \cdot 10^{-16}$ | 11 bits | $2^{-1024} \approx 10^{-308}$ | $2^{1023} \approx 10^{308}$ |
| `long double` | 64 bits | $2^{-64} \approx 5 \cdot 10^{-20}$ | 15 bits | $2^{-16384} \approx 10^{-4932}$ | $2^{16383} \approx 10^{4932}$ |

**Exercice 4.3.** On peut empiriquement tester ces valeurs à l'aide du programme `precision.cc`. Il provoque délibérément une erreur d'arrondi pour déterminer la longueur de la mantisse, ou un dépassement de capacité pour déterminer la plage des exposants. L'idée est de trouver par tâtonnement le plus petit exposant $k > 0$ tel que pour $\varepsilon = 2^{-k}$ on ait `1.0 + eps == 1.0`. Essayer d'expliquer le fonctionnement de ce test, puis l'utiliser pour vérifier le tableau ci-dessus. Vous constaterez de petites différences entre les valeurs du tableau et vos résultats empiriques. (Si cela vous intéresse vous pouvez vous renseigner sur Internet sur la norme IEEE 754 qui définit les nombres flottants dans le moindre détail. Voir aussi `www.vinc17.org/research/papers/arithflottante.pdf`.)

**4.4. Comment calculer avec les flottants ?** À l'instar des opérations réelles $+,-,*,/ : \mathbb{R}\times\mathbb{R}\to\mathbb{R}$ on veut définir les opérations élémentaires `+`, `-`, `*`, `/` pour les nombres flottants. On peut bien-sûr regarder la restriction $+,-,*,/ : R\times R\to\mathbb{R}$, mais dans la majorité des cas les résultats ne retomberont pas dans le sous-ensemble $R\subset\mathbb{R}$ des nombres machine. Il faut donc *arrondir* pour représenter le résultat.

☞ *Due à la précision limitée, les opérations élémentaires ne peuvent rendre qu'une valeur approchée lorsque le résultat exact n'est pas représentable dans la numération choisie. On parle ainsi de l'*arithmétique arrondie *ou aussi de l'*arithmétique flottante. ✍

**Exercice/M 4.4.** Essayez de définir une addition $+ : R\times R\to R$ pour notre exemple minuscule ($\ell+1=3$, $e\in[\![-3,0]\!]$). Quelles additions sont exactes, lesquelles faut-il arrondir ? Dans votre définition vous pouvez vous servir des valeurs exceptionnelles $\pm\infty$ si vous voulez. (Elles sont tellement utiles qu'elles sont prévues dans toutes les implémentations standards des nombres à virgules flottantes.)

☞ On explicitera une implémentation complète au §XVI, quand on parlera du « calcul arrondi fiable ».

Tous les calculs ultérieurs se baseront sur ces opérations élémentaires : évaluation des polynômes ou des fractions rationnels, approximation des fonctions usuelles comme $\sqrt{},\exp,\log,\sin,\cos,\ldots$. Comme les opérations élémentaires sont déjà des approximations, on doit s'attendre à une propagation d'erreurs (parfois non négligeable). On en verra quelques exemples dans la suite.

**4.5. Quand vaut-il mieux calculer de manière exacte ?** Le principal problème du calcul approché est la propagation des erreurs d'arrondi. Ainsi le résultat final calculé peut être assez éloigné du résultat exact cherché. Un exemple flagrant de ce genre est le « polynôme de Rump » (voir `rump.cc`) :

$$f(x,y) = \frac{1335}{4}y^6 + \frac{11}{2}y^8 + \frac{x}{2y} + x^2\left(11x^2y^2 - y^6 - 121y^4 - 2\right)$$

On trouve par un calcul exact que $f(77617,33096) = -54767/66192 \approx -0.8274$. Or, l'évaluation en $x = 77617$, $y = 33096$ provoque de graves problèmes quand on utilise des nombres à virgule flottante :

- Calcul de $f(77617,33096)$ utilisant le type `float` : `-1.1056291e+30`
- Calcul de $f(77617,33096)$ utilisant le type `double` : `1.787028332140613e+20`
- Calcul de $f(77617,33096)$ utilisant le type `long double` : `0`

On observe ici une perte totale de chiffres significatifs autrement dit une explosion de l'erreur relative. On n'arrive même pas à déterminer le bon signe ! Si vous voulez vérifier ces résultats, tout à fait imprévisibles, vous pouvez vous servir du programme `rump.cc`. Par exemple, vous pouvez tester des paramètres voisins afin de comprendre un peu mieux ce qui se passe. (Voir §5.4 pour un phénomène similaire d'annulation.) Ici le calcul exact est clairement préférable : la question admet une formulation qui ne fait intervenir que de nombres rationnels, et les calculs exacts à effectuer sur ordinateur ne sont pas trop coûteux.

## 5. Des pièges à éviter

Le but principal des exercices suivants est de vous convaincre que les nombres machine modélisent assez mal le corps $\mathbb{R}$ des nombres réels ! Ainsi l'usage du calcul numérique puis l'interprétation des résultats présentent des difficultés particulières. Une certaine expérience et un esprit critique sont donc importants pour tout utilisateur, et d'autant plus pour tout programmeur qui se respecte.

Même sans ambitions approfondies, il faut au moins connaître *de bons et de mauvais exemples*. Nous allons donc faire quelques expériences numériques sur ordinateur. Vous trouverez dans la suite des exemples suffisamment drastiques, j'espère, pour vous vacciner durablement contre le calcul naïf. Notre but sera ensuite de comprendre sous quelles conditions un calcul arrondi peut tout de même aboutir à un résultat significatif.

**5.1. Nombres non représentables.** Les erreurs d'arrondi peuvent se produire dans les calculs les plus simples, même avec des nombres rationnels. Voici un exemple typique. Le calcul suivant ne donne pas 0, comme il serait mathématiquement correct. Le tester puis expliquer son résultat :

```
float a= 10.0 / 3.0;        // calcul exact : a vaut 10/3
float b=    a - 3.0;        //                b vaut 1/3
float c=    b * 3.0;        //                c vaut 1
float d=    c - 1.0;        //                d vaut 0
cout << d << endl;          // Que vaut le résultat approximatif ?
```

Effectuer aussi le calcul similaire `a=10.0/4.0; b=a-2.0; c=b*2.0; d=c-1.0;` Cette fois-ci le résultat est-il exact ? Comment expliquer ce phénomène chanceux ?

☞ *Même les calculs les plus innocents peuvent produire des erreurs d'arrondi.* ✍

**5.2. Quand deux nombres flottants sont-ils « égaux » ?** Les erreurs d'arrondi sont aussi inévitables qu'imprévisibles. Pour en comprendre les effets catastrophiques possibles, souvent inattendus, regardons un logiciel de gestion comme le suivant (légèrement simplifié).

**Programme XV.1** L'agent comptable est innocent ! `compte.cc`

```
#include <iostream>
using namespace std;

int main()
{
  float somme= 0.0;
  for ( int i=1; i<=10; ++i ) somme+= 0.1;
  cout << "La somme vaut " << somme << "." << endl;
  if ( somme == 1.0 ) cout << "Le compte est bon." << endl;
  else cout << "Vous êtes accusé de détournement de fonds." << endl;
}
```

**Exercice/P 5.1.** Quel résultat ce programme donne-t-il ? Le tester puis l'expliquer. *Indication.* — On pourra faire afficher `somme-1` pour tester si les deux flottants sont égaux ou seulement très proches. Par curiosité on pourrait augmenter la précision : est-ce que le type `float` ou `double` ou `long double` joue un rôle ? A priori on ne s'attend pas aux erreurs quand on calcule avec des nombres « ronds » comme 0.1. Pour comprendre l'occurrence des erreurs d'arrondi, déterminer le développement binaire de $0,1_{\text{dec}}$.

☞ *Afin de tester l'égalité de deux nombres flottants, il faut remplacer `a==b` par `abs(a-b)<eps` et `a!=b` par `abs(a-b)>=eps`.* ✍

**5.3. Combien de chiffres sont significatifs ?** Le fichier en-tête `<cmath>` fournit quelques fonctions usuelles, comme `exp`, `log`, `pow`, `sqrt`, etc. Quelle précision peut-on attendre de ces fonctions ?

**Exercice/P 5.2.** Pour plus de précision on pourrait être tenté d'afficher plus de chiffres, par exemple :

```
float a= sqrt(2.0); cout.precision(50); cout << a << endl;
```

Expliquer l'erreur logique dans cette approche. Combien de chiffres sont significatifs ? Pour résoudre ce mystère et pour bien rigoler, remplacer `sqrt(2.0)` par `10.0/3.0`, puis `float` par `double`, ...

*Il faut se méfier de l'affichage inutile de chiffres non significatifs.*

Psychologiquement, un nombre affiché avec beaucoup de décimales suggère une grande précision, le problème étant que les chiffres terminaux n'ont souvent aucune signification ! Vous pouvez vérifier cette observation au quotidien. Penser par exemple à un sondage qui parle de « 57,14% des personnes interrogées » au lieu de dire « quatre des sept personnes interrogées ».

*L'affichage des chiffres non significatifs est soit maladroite soit malhonnête.*
*Afficher n décimales seulement si l'erreur relative est plus petite que* $5 \cdot 10^{-n}$.

**5.4. Perte de chiffres significatifs.** D'après ce qui précède, il faut préserver précieusement un maximum de chiffres significatifs, ce qui est souvent difficile. Par contre, les expériences suivantes montrent qu'il est très facile de détruire la signification d'un résultat.

**Exercice/P 5.3.** Incroyable mais vrai : l'addition des flottants n'est même pas associative ! Pour `a=-1e30, b=1e30, c=1.0` comparer `(a+b)+c` et `a+(b+c)`. Expliquer la différence.

*L'addition de deux nombres, l'un « grand » l'autre « petit »,*
*entraîne la perte de chiffres significatifs contribués par le petit.*

**Exercice/P 5.4.** La définition $f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$ pourrait inspirer la tentative d'une dérivation numérique comme suit. Essayez d'en prédire le résultat, puis vérifiez-le.

**Programme XV.2** Dérivation numérique naïve `derivation.cc`

```
1  #include <iostream>
2  using namespace std;
3
4  double f( double x )
5  { return 3.14159265358979323846 * x * x; }
6
7  int main()
8  {
9    cout.precision(20);                  // demander 20 décimales à l'affichage
10   double a= 1.0, eps= 1.0;             // on calcule avec 16 décimales
11   for ( int i=0; i<60; ++i, eps/=2 )
12     cout << ( f(a+eps)-f(a) ) / eps << endl;
13 }
```

*La soustraction de deux nombres proches, ou l'addition de deux nombres presque opposés,*
*produit une perte (parfois considérable voire totale) de chiffres significatifs. À éviter !*

En voici un exemple :

| | |
|---|---|
| $x = 1,23456789047321$ | avec quinze décimales significatives |
| $y = 1,23456789021588$ | avec quinze décimales significatives |
| $x - y = 0,00000000025733$ | seulement cinq décimales significatives |

**5.5. Comment éviter une perte de chiffres significatifs ?** Regardons un exemple important : comment implémenter une approximation pas trop mauvaise de l'exponentielle $\exp\colon \mathbb{R}\to\mathbb{R}_+$ ? Le programme `exp.cc` le fait via la série $\exp(x)=\sum_{k=0}^{\infty}\frac{1}{k!}x^k$. Évidemment on ne peut pas sommer une infinité de termes, on doit donc se contenter d'aller jusqu'à un certain rang $n$. Celui-ci doit être choisi

– suffisamment grand pour garantir une bonne majoration du reste négligé,
– mais pas inutilement grand afin d'obtenir un calcul efficace.

Dans notre exemple naïf on laisse le choix de $x$ et $n$ à l'utilisateur. Ensuite le programme évalue la série tronquée après le $n$ième terme, $s_n(x)=\sum_{k=0}^{n}\frac{x^k}{k!}$, par la méthode de Horner :

$$s_n(x)=\sum_{k=0}^{n}\frac{x^k}{k!}=\left(\left(\dots\left(\left(\left(\frac{x}{n}+1\right)\frac{x}{n-1}+1\right)\frac{x}{n-2}+1\right)\dots\right)\frac{x}{2}+1\right)\frac{x}{1}+1$$

Soulignons qu'il est toujours une bonne idée de considérer Horner quand il s'agit d'évaluer un polynôme : c'est simple et efficace. Pourtant, dans le calcul numérique deux problèmes se posent :

– Si $|x|$ est très grand, les premiers termes $\frac{|x^k|}{k!}$ croissent avant que la factorielle $k!$ ne l'emporte. Ceci veut dire qu'il faut aller assez loin dans la série afin d'obtenir un reste suffisamment petit. Le tester empiriquement pour $x=10,20,30,40,50,\dots$.
– Pour $x<0$ les termes $\frac{x^k}{k!}$ sont de signes alternés. Quand $|x|$ est grand, ceci entraîne une perte dramatique de chiffres significatifs. Le tester empiriquement pour $x=-10,-20,-30,-40,-50,\dots$. Pourquoi ne sert-il plus à rien d'augmenter le rang $n$ dans ce cas ?

☞ *Souvent la perte de précision peut être évitée en reformulant le calcul.* ✍

Dans ce cas concret la solution est très simple : on évalue la série seulement quand $|x|$ est petit, disons pour $x$ dans l'intervalle $[-1,1]$ où le comportement numérique est excellent. Pour $|x|>1$ on se ramène au cas précédent via $\exp(x)=\exp(x/2)^2$, en profitant de notre précieuse connaissance de la fonction exp.

**Exercice/M 5.5.** Montrer l'égalité de $\sqrt{1+x}-1$ et $\frac{x}{1+\sqrt{1+x}}$. Prédire puis comparer les résultats d'un calcul numérique pour $x$ proche de zéro. Quelle formule est préférable et pourquoi ?

**Exercice/M 5.6.** Discuter la formule $\sinh(x)=\frac{1}{2}(e^x-e^{-x})$ sous l'aspect d'une éventuelle perte de précision. Esquisser une implémentation sérieuse de la fonction $\sinh(x)$ pour $x$ proche de 0. Est-ce que le même problème se pose pour $\cosh(x)=\frac{1}{2}(e^x+e^{-x})$ ?

**5.6. Phénomènes de bruit.** Rappelons que la méthode dichotomique pour résoudre une équation réelle $f(x)=0$ se base sur le théorème des valeurs intermédiaires :

**Théorème 5.7.** *Soit $f\colon[a,b]\to\mathbb{R}$ une fonction continue et $y$ une valeur comprise entre $f(a)$ et $f(b)$. Alors il existe $x\in[a,b]$ de sorte que $f(x)=y$. En particulier, si $f(a)$ et $f(b)$ n'ont pas le même signe, alors il existe un nombre réel $x\in[a,b]$ avec $f(x)=0$.*

**Exercice/M 5.8.** La méthode dichotomique pour résoudre $f(x)=0$ procède comme suit : supposons que $a_k<b_k$ et $f(a_k)<0<f(b_k)$. On prend $x_k=\frac{1}{2}(a_k+b_k)$ puis on compare : si $f(x_k)>0$ alors on continue avec l'intervalle $[a_k,x_k]$, si $f(x_k)<0$ alors on continue avec l'intervalle $[x_k,b_k]$. Montrer que cette algorithme produit une suite $(x_k)$ qui converge vers une limite $x\in[a,b]$ vérifiant $f(x)=0$. Montrer de plus la majoration $|x-x_k|\le\frac{1}{2}|b_k-a_k|=2^{-k}|b-a|$, ce qui prouve une convergence linéaire.

Selon vos expériences numériques, quels problèmes prédiriez-vous pour une implémentation de cette méthode sur ordinateur ? Quelle précision peut-on espérer à réaliser ? Quels sont les facteurs limitant ?

Le programme suivant illustre un phénomène de « bruit » très embêtant dans l'évaluation de fonctions, aussi gentilles qu'elles soient. On évalue le polynôme $f(x)=x^6-9x^5+30x^4-40x^3+48x-32$ par la méthode de Horner, ce qui en soi est une bonne idée. Puis on affiche les valeurs $f(x)$ pour $x$ proche de 2.

**Exercice 5.9.** Vérifier que le polynôme $f$ s'annule en $x=2$ (exactement). Numériquement, en utilisant le type `double`, il se trouve que pour $x\in[1,9997;2,0003]$ la valeur $f(x)$ *oscille* aléatoirement autour de 0. Vérifiez-le et essayez d'expliquer ce phénomène. Est-ce une propriété de la fonction $f$ ou bien un artefact de notre implémentation ? Bien sûr, ce phénomène de bruit est désespérant quand on cherche à trouver une solution de $f(x)=0$ par la méthode dichotomique !

**Programme XV.3** Bruit « aléatoire » dans l'évaluation d'une fonction `bruit.cc`

```
#include <iostream>
using namespace std;

double f( const double& x )
{ return ((((x-9)*x+30)*x-40)*x*x+48)*x-32; }

int main()
{
  cout.precision(10);                              // Demande d'afficher 10 décimales
  cout.setf( ios::scientific | ios::showpos );     // pour bien aligner les chiffres.
  for( double x=1.9997; x<=2.0003; x+=0.00001 )    // Cette boucle produit un tableau.
    cout << "f(" << x << ") = " << f(x) << endl; // Il y aura des surprises...!
}
```

**5.7. Conditions d'arrêt.** Un problème particulier se pose pour la condition d'arrêt dans une itération $x_{k+1} = f(x_k)$. Supposons que la suite converge vers une limite $x = \lim x_k$, dont on cherche une valeur approchée à une certaine précision $\varepsilon > 0$ près. Mathématiquement il faut itérer jusqu'à ce que $|x - x_k| \leq \varepsilon$. (Dans la pratique on ne connaît en général pas la valeur $x$, donc on remplace cette condition d'arrêt par $|x_k - x_{k-1}| \leq \varepsilon$, et on majore la vraie distance $|x - x_k|$ par une autre méthode.) Numériquement cette condition peut parfois être impossible à atteindre. Reprenons la méthode de Newton :

**Programme XV.4** Première tentative d'implémenter Newton-Héron

```
float inf= 1, sup= a, ecart;
do {
  sup= ( (n-1)*sup + inf ) / n;
  inf= a / puissance( sup, n-1 );
  ecart= abs( sup-inf );
} while( ecart >= eps );  // Cette condition d'arrêt est-elle raisonnable ?
```

**Exercice/P 5.10.** À cause de la précision limitée, la condition d'arrêt risque de causer de sérieux problèmes. Tout marche, par chance, pour $a = 2$, $n = 2$, $\varepsilon = 1e-20$, mais tourne à la catastrophe pour $a = 3$. Le tester empiriquement ; il faut donc diminuer la précision exigée. Que se passe-t-il quand on remplace `float` par `double` puis par `long double` ? Quelle précision est raisonnable ? Comment le savoir d'avance ?

☞ *Même avec le calcul le plus sophistiqué on ne peut pas surpasser la précision (souvent très limitée) des nombres flottants utilisés.* ✍

Malgré ces difficultés, il faut modifier la condition d'arrêt de sorte que le calcul se termine toujours : soit l'écart souhaité est atteint, soit l'écart ne diminue plus. Tester ainsi le programme `racine.cc`, convenablement modifié. Peut-on ainsi calculer $\sqrt[2]{3}$ à $\varepsilon = 10^{-10}$ près ? à $\varepsilon = 10^{-20}$ près ? Peut-on être sûr de la précision atteinte ? (On développera une réponse satisfaisante avec le calcul fiable plus bas.)

**Programme XV.5** Seconde tentative d'implémenter Newton-Héron

```
float inf= 1, sup= a, ecart= abs( sup-inf ), ecart_precedent;
do {
  sup= ( (n-1)*sup + inf ) / n;
  inf= a / puissance( sup, n-1 );
  ecart_precedent= ecart;
  ecart= abs( sup-inf );
} while( ecart >= eps && ecart < ecart_precedent );
```

**5.8. Équations quadratiques.** L'équation $ax^2+bx+c=0$ admet deux solutions, données par la formule bien connue $x_\pm = \frac{1}{2a}(-b \pm \sqrt{b^2-4ac})$. Outre les opérations arithmétiques elle n'utilise que la fonction $x \mapsto \sqrt{x}$, dont on vient de discuter une méthode d'approximation numérique. Le programme `quadratique.cc` en déduit une implémentation hâtive et insoucieuse pour les équations quadratiques. Il résout correctement $x^2-x-2=0$, mais il traite beaucoup d'autres cas de manière maladroite :

(1) Notre programme ne fait aucun effort pour éviter une perte de chiffres significatifs : Pour $x^2-12345x+1=0$ il trouve $x_+ = 12345$ et $x_- = 0$ au lieu de $x_+ \approx 12344,99992$ et $x_- \approx 0,00008$. L'erreur relative de $x_+$ est petite, mais pour $x_-$ elle est de 100%. Dans un tel cas il sera avantageux de calculer d'abord $x_+$ puis $x_- = \frac{c}{ay}$. (Le cas inverse est également possible.)

(2) Dans le cas $a=0$ il s'agit d'une équation linéaire, ce qui est plus facile mais nécessite un traitement à part. Lancer notre programme pour résoudre $x-2=0$, il y aura des surprises...

(3) Dans le cas $d<0$ les solutions sont complexes ; tester le comportent du programme sur l'exemple $x^2+2x+3$. Pour un programme qui se respecte il sera certes une bonne idée de prévoir des solutions complexes, ou mieux encore d'autoriser même des coefficients complexes.

☞ Un nombre complexe peut être modélisé par une paire de nombres flottants, ce qui entraîne les avantages et inconvénients discutés plus haut : calcul efficace mais erreurs d'arrondi inévitables. Après inclusion du fichier en-tête `<complex>` vous pouvez utiliser les types `complex<float>`, `complex<double>`, etc. Si vous voulez, modifier ainsi le programme `quadratique.cc` et optimisez-le.

## 6. Sommation de séries

Dans les exercices suivants vous pouvez expérimenter avec les différents types de nombres flottants comme `float`, `double`, `long double`, ou bien `mpf_class` de la bibliothèque GMP (tapez `info gmp C++` dans une ligne de commande).

**Exercice/P 6.1.** La série $\sum \frac{1}{k}$ diverge, c'est-à-dire les sommes partielles $s_n = \sum_{k=1}^{n} \frac{1}{k}$ croissent sans borne. Pourtant elles deviennent stationnaires quand on les calcule naïvement sur ordinateur ! Essayez de prédire la valeur stationnaire pour le type `float`. Le vérifier avec le programme `divergence.cc`. Quelle valeur trouvez-vous ? Que se passe-t-il avec le type `double` ?

☞ *Se méfier d'une apparente « convergence numérique » sans contrôle d'erreur.* ✍

**Exercice/P 6.2.** La série $\sum_{k=1}^{\infty} \frac{1}{k^2}$ converge. Pour approcher la limite, en utilisant le type `float`, calculer $\sum_1^n \frac{1}{k^2}$ dans le sens des indices croissants, puis $\sum_n^1 \frac{1}{k^2}$ dans le sens des indices décroissants. Vu la nature des erreurs d'arrondi, quelle approche vous semble plus exacte ? Comparer avec les résultats obtenus avec le type `double` et `long double`.

☞ *Lors d'une sommation numérique l'ordre des termes peut influencer le résultat.* ✍

**Exercice/M 6.3.** Afin d'encadrer la valeur de $\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2}$ il ne suffit évidemment pas de calculer la somme partielle $s_n$, il faut aussi majorer le reste $r_n = \sum_{k=n+1}^{\infty} \frac{1}{k^2}$. Montrer que $\frac{1}{n+1} < r_n < \frac{1}{n}$ :

(1) via l'encadrement $\frac{1}{k} - \frac{1}{k+1} < \frac{1}{k^2} < \frac{1}{k-1} - \frac{1}{k}$ puis une somme télescopique,

(2) via l'encadrement $\int_k^{k+1} \frac{1}{x^2}dx < \frac{1}{k^2} < \int_{k-1}^{k} \frac{1}{x^2}dx$ puis la somme des intégrales.

En déduire un programme qui calcule un encadrement de $\zeta(2)$ en fonction de $n$. (On sait d'ailleurs que $\zeta(2) = \pi^2/6$, mais ce beau résultat ne joue pas de rôle ici.)

☞ *C'est la majoration du reste qui fait d'une série $\sum_{k=1}^{\infty} a_k$ une méthode praticable pour calculer une valeur approchée de la somme.* ✍

**Exercice/M 6.4.** En généralisant l'exercice précédent, écrire un programme qui calcule un encadrement de $\zeta(3) = \sum_{k=1}^{\infty} \frac{1}{k^3}$ en fonction de $n$. Si vous voulez, vous pouvez étendre cette approche afin d'encadrer $\zeta(s)$ pour $s = 2,3,4,5,\ldots$. Est-ce envisageable pour tout $s>1$ ?

☞ *Essayez toujours de justifier vos résultats numériques en précisant leur marge d'erreur.* ✍

**Exercice/P 6.5.** Afin de calculer l'exponentielle $\exp(x)$ on pourrait théoriquement se servir de la limite $\exp(x) = \lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n$. Quels problèmes prédiriez-vous ? Pour le tester empiriquement, implémentez efficacement ce calcul et essayez d'ainsi calculer $\exp(\pm 1)$ ou $\exp(\pm 100)$ à 10 décimales près.
*Indication.* — Il est inutile de parcourir tous les cas $n = 1,2,3,\ldots$, vous pouvez en choisir une suite extraite judicieuse. À noter que pour $n = 2^k$ la puissance $a^n$ est particulièrement facile à calculer : évitez une boucle $a^1, a^2, a^3, \ldots, a^n$ de longueur $n$, une boucle $a^2, a^4, a^8, \ldots, a^n$ de longueur $k$ suffit !

Comparer avec l'approche du §5.5. Si vous voulez, vous pouvez complétez le programme `exp.cc` en une implémentation plus robuste qui calcule l'exponentielle à au moins 10 décimales près. Le tester sur beaucoup de valeurs de $x$, petites et grandes, positives et négatives.

**Exercice/P 6.6.** Afin de calculer $\ln 2$ on pourrait théoriquement se servir de la série $\ln 2 = \sum_{k=1}^{\infty}(-1)^{k+1}\frac{1}{k}$. Écrire un programme qui calcule $s_n = \sum_{k=1}^{n}(-1)^k\frac{1}{k}$. Majorer le reste $r_n$ puis donner un encadrement de $\ln 2$. Peut-on ainsi calculer $\ln 2$ à $10^{-6}$ près ? à $10^{-12}$ près ?

☞ *On a tout intérêt à choisir, si possible, une série qui converge rapidement.* ✍

**Exercice/P 6.7.** Dans l'exemple précédent, rien ne nous empêche de calculer $\ln 2$ par une série mieux adaptée. Pour $|x| < 1$ on a les développements en série

$$\ln(1+x) = +x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots$$
$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \ldots$$
$$\Longrightarrow \quad \ln\left(\frac{1+x}{1-x}\right) = 2\left(x + \frac{x^3}{3} + \frac{x^5}{5} + \ldots\right)$$

Pour $x = \frac{1}{3}$ on obtient ainsi $\ln 2 = \sum_{k=1}^{\infty}\frac{2}{2k-1}3^{1-2k}$. Écrire un programme qui calcule la somme partielle $s_n = \sum_{k=1}^{n}\frac{2}{2k-1}3^{1-2k}$. Majorer le reste $r_n$ puis donner des encadrements de $\ln 2$. Peut-on ainsi calculer $\ln 2$ à $10^{-10}$ près ? à $10^{-100}$ près ?

*Numerical data piles up and numerical programs grow ever more ambitious and complicated while their users become, on average, far less knowlegeable about numerical error-analysis, though no less clever than their predecessors about subjects they care to learn. Consequently numerical anomalies go mostly unobserved or, if observed, routinely misdiagnosed. Fortunately most of them don't matter. Most computations don't matter.*
W. Kahan, *How futile are mindless assessments of roundoff in floating-point computation?*

PROJET XV

# Dérivation numérique et extrapolation de Richardson

Avant de traiter l'intégration numérique, le présent projet discute la dérivation numérique, souvent plus facile. Le développement qui suit introduit une technique fondamentale : l'extrapolation de Richardson. Elle nous servira plus tard pour l'intégration dans la méthode de Romberg.

Les expériences numériques du chapitre XV (exercice 5.4) ont déjà montré que le calcul numérique d'une dérivée $\lim_{h\to 0} \frac{f(a+h)-f(a)}{h}$ est loin d'être trivial : si les valeurs $f(a)$ et $f(a+h)$ sont calculées avec $n$ chiffres significatifs, il est assez difficile d'en déduire une valeur approchée de $f'(a)$ avec $n$ chiffres significatifs. Dans de telles situations, l'extrapolation de Richardson peut remédier à la perte de précision.

**Sommaire**



## 1. Convergence linéaire vs quadratique

Nous supposons que $f : ]a-\varepsilon, a+\varepsilon[ \to \mathbb{R}$ est de classe $C^{n+1}$, avec $n$ aussi grand que nécessaire. Nous avons en particulier la formule de Taylor (avec reste de Lagrange) :

$$f(a+h) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!} h^k + \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1} \quad \text{avec} \quad \xi = a+\theta h \text{ et } \theta \in ]0,1[.$$

Du coté numérique, nous disposons d'une implémentation `Flo f( Flo x )` qui calcule $f$ avec une précision satisfaisante, disons de 15 décimales, en utilisant un type flottant que l'on nommera `Flo`. À noter toutefois qu'un tel calcul peut être coûteux ; on essaiera de s'en servir avec modération.

Par contre, nous n'avons aucune connaissance des dérivées $f', f'', \ldots$, on suppose seulement leur existence. Notre but est de développer une méthode efficace pour calculer une valeur approchée de $f'(a)$ à partir de très peu de valeurs de $f$ dans un voisinage de $a$.

**Exercice/M 1.1.** Dans un premier temps, on pourrait prendre $\phi_1(h) := \frac{f(a+h)-f(a)}{h}$ comme valeur approchée de la dérivée exacte $f'(a)$. Vérifier l'estimation d'erreur $\phi_1(h) = f'(a) + \frac{1}{2} f''(\xi) h$. Pour $h \to 0$ la convergence $\phi_1(h) \to f'(a)$ est donc au moins *linéaire*, souvent abrégé $\phi_1(h) = f'(a) + O(h)$.

Un peu plus raffinée, considérons la valeur approchée $\phi_2(h) := \frac{f(a+h)-f(a-h)}{2h}$. Vérifier l'estimation d'erreur $\phi_2(h) = f'(a) + \frac{1}{3!} f'''(\xi) h^2$. Pour $h \to 0$ la convergence $\phi_2(h) \to f'(a)$ est donc au moins *quadratique*, souvent abrégé $\phi_1(h) = f'(a) + O(h^2)$. Ceci explique l'intérêt de cette deuxième approche.

**Exercice/P 1.2.** Comparer les deux approches sur un exemple numérique, disons $f : \mathbb{R} \to \mathbb{R}$ donnée par $f(x) = \sin(x)$. Calculer des valeurs approchées de $f'$ en $a = 1$ par les deux méthodes ci-dessus pour $h = 2^{-k}$ avec $k = 1, 2, 3, \ldots, 50$. On pourra commencer l'implémentation par

```
typedef double Flo;
const Flo a= 1.0;
```

Bien sûr on connaît la valeur $f'(a) = \cos(1) \approx 0.54030230586$ ; pour information faites afficher `cos(a)` de la bibliothèque `cmath`. Quelle est la précision maximale que l'on puisse atteindre avec la méthode linéaire ? avec la méthode quadratique ? Quelles sont les valeurs optimales pour $k$, environ ? Expliquer pourquoi il faut choisir $k$ ni trop petit ni trop grand.

## 2. Convergence d'ordre 4

**Exercice/M 2.1.** Nous nous proposons d'obtenir une convergence encore plus rapide. Reprenons la dérivée approchée $\phi_2(h) = \frac{f(a+h)-f(a-h)}{2h}$. En supposant $f$ de classe $C^{2n+3}$, montrer que

$$\phi_2(h) = f'(a) + a_2h^2 + a_4h^4 + \dots a_{2n}h^{2n} + \frac{f^{(2n+3)}(\xi)}{(2n+3)!}h^{2n+2}.$$

Vérifier que pour tout $\alpha \in \mathbb{R}$ la fonction $\phi_4(h) := \phi_2(h) + \alpha\big[\phi_2(h) - \phi_2(2h)\big]$ converge vers $f'(a)$ pour $h \to 0$. Déterminer $\alpha$ de sorte que la convergence soit d'ordre 4.

**Exercice/P 2.2.** Implémenter la méthode ci-dessus pour notre exemple $f(x) = \sin(x)$ et calculer ainsi des valeurs approchées de $f'$ en $a = 1$ en calculant $\phi_2(h)$ puis $\phi_4(h)$ pour pour $h = 2^{-k}$ avec $k = 1, 2, 3, \dots, 50$. Veillez à ne pas évaluer la fonction $f$ inutilement, en réutilisant les valeurs de $\phi_2$ déjà calculées. Quelle est la précision maximale que l'on puisse atteindre ? Quelle est la valeur optimale pour $k$, environ ? En quoi la méthode est-elle intéressante ?

## 3. Extrapolation de Richardson

En généralisant ce qui précède, supposons que $\phi\colon ]0,\varepsilon] \to \mathbb{R}$ est une fonction que l'on sait calculer pour $\varepsilon 2^{-k}$ avec $k = 0, 1, 2, \dots$. Afin d'en déduire une valeur approchée de $\lim_{h\to 0}\phi(h)$, on supposera que $\phi$ admet un développement limité

$$\phi(h) = a_0 + a_2h^2 + a_4h^4 + \dots + a_{2n}h^{2n} + O(h^{2n+2})$$

avec des constantes $a_0, a_2, a_4, \dots, a_{2n} \in \mathbb{R}$ que nous ignorons. Pour $k = 0, 1, 2, \dots$ on pose $d_k^0 := \phi(\varepsilon 2^{-k})$ ; c'est la première ligne du schéma suivant. La convergence $d_k^0 \to \phi(0)$ est quadratique. Afin d'obtenir une convergence plus rapide, on calcule la ligne $i = 1, 2, \dots, n$ par $d_k^i := d_{k+1}^{i-1} + \frac{1}{4^i-1}\big[d_{k+1}^{i-1} - d_k^{i-1}\big]$.

$$\begin{array}{cccccc}
d_0^0 & d_1^0 & d_2^0 & \dots & d_n^0 & \to \ \phi(0) \\
d_0^1 & d_1^1 & \dots & d_{n-1}^1 & & \to \ \phi(0) \\
d_0^2 & \dots & d_{n-2}^2 & & & \to \ \phi(0) \\
\vdots & & & & & \\
d_0^n & & & & & \to \ \phi(0)
\end{array}$$

**Exercice/M 3.1.** Vérifier que $d_k^i \to \phi(0)$ pour $k \to \infty$ et déterminer l'ordre de la convergence.

**Exercice/P 3.2.** Nous supposons de disposer d'une implémentation `Flo phi( Flo h )`. Écrire une fonction `Flo richardson( Flo eps, int n )` qui met en œuvre l'algorithme développé ci-dessus pour calculer $\lim_{h\to 0}\phi(h)$.
*Indication.* — Une méthode éprouvée est de ne stocker que la diagonale $d_0^k, \dots, d_{k-2}^2, d_{k-1}^1, d_k^0$. Explicitement : pour $k = 0$ on calcule $d_0^0$ et le stocke dans un vecteur ; pour $k = 1$ on ajoute $d_1^0$ et remplace $d_0^0$ par $d_0^1$ ; pour $k = 2$ on ajoute $d_2^0$, remplace $d_1^0$ par $d_1^1$, puis $d_0^1$ par $d_0^2$. Ainsi le vecteur s'agrandit chaque fois d'un élément, les améliorations se propagent de la fin vers le début, et la « meilleur approximation » $d_0^k$ est toujours stockée en position 0. C'est la valeur $d_0^n$ qui est finalement renvoyée par la fonction. (Pour des tests vous pouvez faire afficher $d_0^k$ dans chaque itération.)

**Exercice/P 3.3.** Appliquer votre fonction `richardson` à l'exemple $f(x) = \sin(x)$ et calculer ainsi des valeurs approchées de $f'$ en $a = 1$ pour $\varepsilon = 1$ et $n = 1, \dots, 20$. Peut-on choisir $n$ trop petit ? trop grand ? Quelles sont les valeurs acceptables pour $n$, environ ? Cette méthode vous semble-t-elle assez robuste ?

**Exercice 3.4.** Vérifier que $\pi = \lim_{k\to\infty} 2^k \sin(2^{-k}\pi)$. Interpréter géométriquement ces valeurs en comparant avec la circonférence d'un $2^k$-gone régulier inscrit dans un cercle de rayon 1. Pour calculer $a_k = \sin(2^{-k}\pi)$ numériquement on n'utilisera pas de valeur approchée de $\pi$, ni la fonction sin, mais la formule de récurrence $1 - 2a_{k+1}^2 = \sqrt{1 - a_k^2}$ en commençant par $a_1 = 1$. (La vérifier.) Calculer ainsi des valeurs approchées de $2^k \sin(2^{-k}\pi)$ pour $k = 1, 2, 3, \dots$ et juger si la convergence est satisfaisante. Pourquoi cette formule de récurrence est-elle mal adaptée au calcul numérique ? À quelle précision pouvez-vous ainsi approcher $\pi$ ? Appliquer la méthode de Richardson pour calculer une meilleur approximation de $\pi$.