# Évaluation efficace de fonctions numériques - Outils et exemples

Sylvain Chevillard

N° d'ordre : 519 N° attribué par la bibliothèque : 07ENSL0 519

Laboratoire de l'informatique et du parallélisme

Thèse
en vue d'obtenir le grade de

docteur de l'Université de Lyon - École normale supérieure de Lyon
spécialité informatique
école doctorale *informatique et mathématiques* (InfoMaths)

---

# Évaluation efficace de fonctions numériques

# Outils et exemples

---

présentée et soutenue publiquement par

Sylvain Chevillard

le 6 juillet 2009

| | |
|---|---|
| directeur de thèse | Jean-Michel Muller |
| co-directeur de thèse | Nicolas Brisebarre |
| après avis de | Bruno Salvy, membre/rapporteur |
| | Peter Tang, membre/rapporteur |

devant la commission d'examen formée de

Jean-Paul Allouche, membre
Bernhard Beckermann, membre
Nicolas Brisebarre, membre
Jean-Michel Muller, membre
Bruno Salvy, membre/rapporteur
Peter Tang, membre/rapporteur

# Évaluation efficace de fonctions numériques
## Outils et exemples

Laboratoire de l'informatique du parallélisme
Unité mixte CNRS - ENS Lyon - INRIA - UCB Lyon n° 5668
École normale supérieure de Lyon
46 allée d'Italie, 69 364 Lyon Cedex 07 France

*À Jean-Yves*

*Ce document est une version corrigée, en date du 16 mars 2022.*

Le manuscrit, dans sa version officielle définitive (tel qu'il est déposé en bibliothèque par exemple) est accessible en ligne à l'adresse

https://hal.archives-ouvertes.fr/docs/00/46/07/76/PDF/TheseSylvainChevillard.pdf

Voici la liste des corrections effectuées depuis la version officielle. Les numéros de pages font référence au document officiel, suivis entre parenthèses des numéros de pages dans le présent document :

- page 78 (page 80), en milieu de page, $\frac{x}{2}$ a été remplacé par $\frac{x^2}{2}$ ;
- page 91 (page 93), les occurrences de $x$ ont été remplacées par $x^\star$ dans le troisième point de l'algorithme d'échange ;
- page 92 (page 94), en début de page, correction d'une faute d'orthographe (*formellement identiques* remplacé par *formellement identique*) ;
- dans tout le document, unification des pointeurs vers les différents portails du service HAL : prunel.ccsd.cnrs.fr, hal.inria.fr, tel.archives-ouvertes.fr ont été remplacés par l'adresse unique et plus pérenne hal.archives-ouvertes.fr ;
- page 119 (page 121), en fin de page, $b^\star_{k-1}$, $b^\star_{k-2}$ et $b^\star_1$ ont été remplacés, respectivement, par $b_{k-1}$, $b_{k-2}$, et $b_1$ ;
- page 120 (page 122), en milieu de page, deux occurrences de $b^\star_j$ ont été remplacées par $b_j$ ; en outre, une égalité intermédiaire rappelant que $\langle b_j, b^\star_j\rangle = \langle b^\star_j, b^\star_j\rangle$ a été ajoutée ;
- page 122 (page 124), en fin de page, $\|b^\star_p\|$ a été remplacé par $\|b^\star_p\|^2$.

# Remerciements

Je voudrais commencer par remercier Nicolas. Durant ces trois années, mon encadrement a toujours été sa priorité. Il a donné son temps sans compter, sa disponibilité a été totale, il a toujours répondu présent pour m'aider lorsque le besoin s'en faisait sentir. Tous ses conseils étaient pertinents et cette thèse lui doit beaucoup. Nos nombreuses discussions scientifiques et extra-scientifiques ont été pour moi un soutien et une stimulation.

Nicolas a mené la barque en assumant de façon autonome la direction de cette thèse ; mais l'expérience de Jean-Michel nous a aidés à l'emmener à bon port. Quoiqu'en retrait, il a su intervenir lorsque c'était nécessaire, jalonnant la route de recommandations. Lorsque je le sollicitais — souvent à la dernière minute — pour relire mon travail, il s'y est attelé consciencieusement, avec efficacité. Son point de vue est toujours pertinent.

Bruno et Peter m'ont fait l'honneur de relire mon manuscrit. Se plonger profondément dans le travail de quelqu'un est une tâche difficile. Ils ont accepté de le faire, et la précision de leurs questions et remarques démontre qu'ils y ont apporté beaucoup de soin. Je les remercie pour la qualité de leur travail de lecture, malgré les très brefs délais que je leur ai imposés. Pour Peter, l'exercice était particulièrement difficile du fait de la barrière de la langue. Je l'en remercie d'autant plus. Merci aussi à Bernd et Jean-Paul d'avoir accepté de faire partie de mon jury.

Mes remerciements vont ensuite à Christoph. Christoph est à la fois un camarade de promotion, un collègue, un co-bureau, un co-auteur, un ami. Travailler avec lui a été un véritable plaisir. Nos divergences de points de vue ont été nombreuses, mais toujours constructives. J'ai beaucoup appris à son contact. Sans Christoph et sa ténacité, Sollya n'existerait pas. Notre complicité m'est précieuse ; merci beaucoup pour tout ce que tu m'as apporté. J'espère continuer de travailler avec toi longtemps.

Deux autres personnes ont été fortement impliquées dans cette thèse. Pendant ces trois années, Serge a développé des outils — des prototypes, un peu bricolés, mais toujours d'excellente qualité — qui m'ont beaucoup servi. Il a suivi mon travail, s'est toujours montré très intéressé. Je l'apprécie pour sa bonne humeur, pour sa disponibilité, pour son goût du travail bien fait. Son sens du devoir collectif est inestimable. Mioara a débuté sa thèse alors que la mienne n'était pas loin de s'achever. Néanmoins, nous avons eu le temps de faire un bout de route ensemble. J'espère bien que le chemin ne s'arrêtera pas là.

Jacqueline a relu avec beaucoup d'attention ce manuscrit. La minutie de ses remarques est d'autant plus appréciable que le langage mathématique ne lui est pas familier ; une bonne partie du manuscrit lui était parfaitement incompréhensible. Je ne saurais trop la remercier.

Bien d'autres personnes méritent des remerciements. J'ai eu des interactions nombreuses et enrichissantes avec d'autres étudiants de l'équipe Arénaire : Adrien, Christophe, Guillaume Melquiond, Guillaume Revy, Jérémie. Tous les membres de l'équipe méritent également d'être remerciés pour leur attention et leurs conseils. Les assistantes effectuent un travail admirable ; merci à Corinne, Isabelle, Marie et Sylvie. Les Florents et Julien n'ont contribué à cette thèse en aucune manière ; mais notre amitié et nos discussions (scientifiques ou non) alimentent ma pensée au quotidien. Je

remercie toutes les personnes qui, à un moment ou un autre, m'ont apporté un coup de pouce en me suggérant des idées, en me proposant des pistes de recherche, en relisant mon travail ou en répondant à mes questions.

Je profite de cette occasion pour remercier monsieur Teller. Il a été mon professeur de mathématiques en première année de classes préparatoires au lycée Charlemagne à Paris. J'aimerais pouvoir devenir un aussi bon enseignant.

Pour finir, je remercie Sophie. Sa patience infinie et son soutien pendant les longs mois de rédaction m'ont été précieux. Sa présence au quotidien est importante. Merci pour ton enthousiasme devant les questions informatiques ; merci d'assumer le rôle, pas tous les jours facile, de femme de chercheur.

# Table des matières

# Liste des figures

# Liste des algorithmes

# Notations

Voici quelques notations couramment utilisées dans le manuscrit :

| | |
|---|---|
| $\lfloor x \rfloor$ | Partie entière de $x$ (le plus grand entier inférieur ou égal à $x$). |
| $\lceil x \rceil$ | Partie entière supérieure de $x$ (le plus petit entier supérieur ou égal à $x$). |
| $\lfloor x \rceil$ | Entier le plus proche de $x$. Si $x$ est le milieu de deux entiers, il peut s'agir de n'importe lequel des deux, sauf si une règle est explicitement précisée. |
| $\mathbb{N}$ | Ensemble des entiers naturels $\{0, 1, 2, \ldots\}$. |
| $\mathbb{N}^\star$ | Les entiers, privés de 0 : $\{1, 2, \ldots\}$. |
| $\mathbb{Z}$ | Ensemble des entiers relatifs $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. |
| $\mathbb{R}$ | Ensemble des nombres réels. |
| $\mathcal{F}_t$ | Ensemble des nombres flottants en précision $t$. |
| $\Delta(x)$ | Arrondi de $x$ par excès : le plus petit nombre flottant supérieur ou égal à $x$. |
| $\nabla(x)$ | Arrondi de $x$ par défaut : le plus grand nombre flottant inférieur ou égal à $x$. |
| $\diamond(x)$ | Arrondi fidèle de $x$ : $\diamond(x) \in \{\nabla(x), \Delta(x)\}$. |
| $\circ(x)$ | Arrondi au nombre flottant le plus proche. |
| $\widehat{x}$ | Une valeur approchée du nombre $x$. |
| $\langle k \rangle$ | Accumulation de $k$ erreurs d'arrondi (cf. définition 1.3 page 43). |
| $[a, b]$ | Intervalle fermé, de bornes $a$ et $b$ : $\{x \in \mathbb{R},\ a \leq x \leq b\}$. |
| $[a, b[$ | Intervalle semi-ouvert, de bornes $a$ et $b$ : $\{x \in \mathbb{R},\ a \leq x < b\}$. |
| $[\![a, b]\!]$ | Ensemble des entiers compris entre $a$ et $b$ : $[a, b] \cap \mathbb{Z}$. |
| $\mathrm{sgn}(x)$ | Signe de $x$ : vaut 1 si $x > 0$, $-1$ si $x < 0$ et est indéfini si $x = 0$. |
| $\mathcal{C}^\infty(I)$ | Ensemble des fonctions infiniment dérivables en tout point de l'intervalle $I$. Si le contexte permet de déterminer $I$ sans ambiguïté, on utilise la notation $\mathcal{C}^\infty$. |
| $f(I)$ | Lorsque $f$ est une fonction et $I$ un intervalle, $f(I)$ désigne l'image exacte de $f$ sur l'intervalle $I$ : $f(I) = \{f(x),\ x \in I\}$. |
| $\boldsymbol{r}$ | Un symbole en gras désigne un intervalle. |
| $\underline{\boldsymbol{r}}$ | Borne inférieure de l'intervalle $\boldsymbol{r}$. |
| $\overline{\boldsymbol{r}}$ | Borne supérieure de l'intervalle $\boldsymbol{r}$. |
| $\mathrm{diam}(\boldsymbol{r})$ | Diamètre de $\boldsymbol{r}$ : $\mathrm{diam}(\boldsymbol{r}) = \overline{\boldsymbol{r}} - \underline{\boldsymbol{r}}$. |
| $\|g\|_\infty^{\boldsymbol{r}}$ | Norme sup (ou encore norme infini, ou norme de la convergence uniforme) de $g$ sur $\boldsymbol{r}$ : $\|g\|_\infty^{\boldsymbol{r}} = \max_{x \in \boldsymbol{r}} \lvert g(x) \rvert$. Lorsque le contexte permet de déterminer $\boldsymbol{r}$ sans ambiguïté, on utilise la notation $\|g\|_\infty$. |
| $\langle x, y \rangle$ | Produit scalaire des vecteurs $x$ et $y$. |
| ${}^t A$ | Transposée de la matrice $A$. |
| $\mathcal{O}(g)$ | Ensemble des fonctions $h$ vérifiant $\lvert h \rvert \leq M \lvert g \rvert$, pour une certaine constante $M$. |

# Introduction

On prête à Alston Householder la citation suivante : *It makes me nervous to fly on airplanes, since I know they are designed using floating-point arithmetic.* [1] L'origine de cette citation n'est pas clairement établie ; il s'agissait vraisemblablement d'une critique de Householder à l'encontre du modèle même de virgule flottante. Il semble qu'il ait révisé son jugement par la suite. [2] Je voudrais reprendre cette citation à mon compte — probablement dans un sens qui n'est pas celui que lui donnait Householder. Beaucoup de gens utilisent la virgule flottante, peut-être sans même le savoir, lorsqu'ils effectuent des calculs scientifiques avec un ordinateur. Les nombres flottants constituent une bonne approximation des nombres réels ; on peut légitimement croire qu'il est possible de calculer avec comme s'il s'agissait effectivement de nombres réels exacts. Pourtant ce ne sont que des approximations, et cette subtilité n'est pas sans conséquence. On aurait bien tort de faire une confiance aveugle aux calculs réalisés par ordinateur.

Observons l'exemple suivant qui, bien qu'artificiel, illustre le genre de mésaventure qui pourrait arriver à l'utilisateur naïf :

```
#include <stdio.h>
#include <math.h>

int main(void) {
  double x = 1e-5;

  printf("Lorsque x=%e, (1-x^2/2)-cos(x) = %e.\n", x, (1-x*x/2) - cos(x)  );

  return 0;
}
```

Une fois compilé et exécuté sur mon ordinateur (Pentium D, tournant sous Linux 2.6.26-1 sous Debian, avec gcc 4.3.3 sans option d'optimisation), ce programme écrit en langage C affiche le résultat suivant :

```
Lorsque x=1.000000e-05, (1-x^2/2)-cos(x) = -4.119968e-18.
```

L'utilisateur est content : il a son résultat avec sept chiffres significatifs. Il serait troublé de constater le comportement de sa machine sur l'exemple suivant :

1. Je suis inquiet à l'idée de prendre l'avion depuis que je sais qu'ils sont conçus en utilisant l'arithmétique flottante.

2. L'information la plus fiable que j'aie pu trouver sur le sujet provient d'un billet de Stephen Vavasis, posté sur la liste de diffusion NA-Net le 11 octobre 1997, qui tente de mettre au clair l'histoire de cette citation.

```
#include <stdio.h>
#include <math.h>

int main(void) {
  double x = 1e-5;

  printf("Lorsque x=%e, (1-x^2/2)-cos(x) = %e.\n", x, (1-x*x/2) - cos(x)  );

  printf("Lorsque x=%e, cos(x)-(1-x^2/2) = %e.\n", x, cos(x) - (1-x*x/2)  );
  printf("Lorsque x=%e, cos(x)-1+x^2/2 = %e.\n", x,   cos(x) - 1 + x*x/2  );

  return 0;
}
```

Cette fois, le message affiché est

```
Lorsque x=1.000000e-05, (1-x^2/2)-cos(x) = -4.119968e-18.
Lorsque x=1.000000e-05, cos(x)-(1-x^2/2) = 0.000000e+00.
Lorsque x=1.000000e-05, cos(x)-1+x^2/2 = -1.705029e-20.
```

Notre utilisateur était bien naïf de faire confiance à sa machine puisque la véritable valeur de $(1 - x^2/2) - \cos(x)$ lorsque $x = 1\,\mathrm{e}{-5}$ est en fait $-4,1666666666527\cdots\,\mathrm{e}{-22}$.

Cet exemple montre que, lorsqu'on utilise une arithmétique approchée (comme c'est le cas de l'arithmétique flottante), on n'est jamais assez prudent. Sans doute y a-t-il lieu d'être effectivement inquiet à l'idée que l'arithmétique flottante est utilisée partout, et peut-être par des gens qui n'ont pas conscience de ce genre de danger.

## Petite histoire de l'arithmétique flottante

L'exemple précédent est troublant à plusieurs points de vue : d'abord, bien sûr, parce que les résultats affichés sont complètement faux, mais aussi parce qu'ils sont en contradiction brutale avec des propriétés mathématiques essentielles : par exemple l'associativité et (au moins en apparence) la commutativité de l'addition. Par ailleurs, on peut être troublé que l'ordinateur affiche fièrement sept décimales d'un résultat qui, en réalité, est totalement erroné. Pourtant, il n'y a pas de bug dans le programme ou le compilateur. L'ordinateur a scrupuleusement suivi les règles de calculs de la norme IEEE-754 qui standardise l'arithmétique flottante. On observe ici uniquement les effets dus à la précision limitée des formats de virgule flottante.

L'arithmétique des ordinateurs est la branche de l'informatique qui s'occupe d'étudier et d'améliorer les calculs effectués par les ordinateurs, en particulier les calculs effectués en virgule flottante. Elle permet de comprendre et d'expliquer notre exemple introductif ; elle permet de prédire et éventuellement d'éviter les phénomènes numériques aberrants comme ceux que nous venons de voir. L'équipe Arénaire du laboratoire de l'informatique du parallélisme (LIP) à Lyon s'intéresse à ce type de problématiques depuis une dizaine d'années.

Un grand projet fédérateur de l'équipe Arénaire a été le développement d'une bibliothèque logicielle appelée `CRlibm`. La plupart des travaux présentés dans cette thèse trouvent leur origine dans des problématiques liées au développement de `CRlibm`. Nous présenterons des algorithmes qui permettent d'améliorer la fiabilité, la précision ou la rapidité des approximations utilisées dans `CRlibm`. Le développement de `CRlibm` a été la motivation initiale pour la création de ces algorithmes mais ils ont une portée bien plus large que cette seule bibliothèque : ils s'appliquent, plus généralement,

à n'importe quel contexte où l'on souhaite évaluer des fonctions numériques tout en garantissant que les erreurs ne dépassent pas un certain seuil.

Pendant cette thèse, nous avons écrit un logiciel appelé Sollya. Là encore, le but initial était de donner aux développeurs de `CRlibm` un cadre de travail intuitif, ergonomique et sûr, mais Sollya s'est élargi et a vocation à devenir une véritable boîte à outils fiable pour tous les numériciens.

Pour comprendre les problématiques liées à `CRlibm`, il faut en présenter la genèse et revenir d'abord un peu sur l'histoire de l'arithmétique flottante.

## Naissance de la norme

Dès les premiers calculateurs, le modèle de la virgule flottante s'est imposé pour calculer de manière approchée avec les nombres réels. Coder les nombres en virgule flottante revient à utiliser la notation scientifique que nous connaissons tous : un nombre $x$ est donné par les premiers chiffres significatifs de son développement binaire et une puissance de 2 qui indique son ordre de grandeur. Par exemple, si on prend une précision de 7 bits (c'est-à-dire qu'on ne garde que 7 chiffres binaires significatifs), 42 est codé exactement sous la forme $1,010100_2 \cdot 2^5$, mais 289 (dont le développement binaire exact est $100100001_2$) ne peut être exactement représenté et devra donc être arrondi, par exemple à la valeur $1,001000_2 \cdot 2^8$. La puissance de 2 qui fixe l'ordre de grandeur (5 et 8 dans les exemples précédents) est appelée *l'exposant* du nombre.

Pour calculer avec des nombres en virgule flottante, on doit donc commencer par définir la précision avec laquelle on veut travailler et le nombre de bits sur lequel on va coder l'exposant. La base 2 est un choix naturel puisque les ordinateurs manipulent des données binaires ; cependant, l'utilisation d'autres bases est possible et des implémentations ont existé et existent toujours, qui manipulent des nombres flottants en base 3, 4, 10 ou 16. Les premières machines à calculer électroniques avaient chacune leurs propres caractéristiques : la base de numération, la précision, le nombre de bits sur lequel était stocké l'exposant étaient différents d'un modèle à l'autre.

De plus, même les opérations les plus simples comme l'addition étaient effectuées avec des algorithmes différents sur chaque machine. L'opérateur d'addition renvoyait une valeur approchée du résultat exact, sans autre indication quant à la qualité de cette valeur approchée. Le résultat n'était pas nécessairement le même d'une machine à l'autre. De plus, les cas spéciaux tels qu'une division par 0 ou un dépassement de capacité étaient traités de façon *ad hoc* sur chaque machine : certaines continuaient le calcul comme si de rien n'était, d'autres au contraire s'arrêtaient net avec un message d'erreur, d'autres encore utilisaient un système de drapeaux pour signaler la levée d'une exception.

Cette situation semblait convenir dans un premier temps : puisque, de toutes façons, l'arithmétique flottante était intrinsèquement inexacte, quel mal pouvait-il y avoir à renvoyer tel résultat approché plutôt que tel autre du moment que l'erreur commise restait petite ?

Le problème majeur, dans ces conditions, était la portabilité du code. Un programme pouvait fonctionner parfaitement sur une machine et, sur une autre, s'arrêter brusquement avec une erreur ; la maintenance du code lors d'un changement d'architecture était une tâche de titan. Aussi, vers la fin des années 70, l'idée de standardiser l'arithmétique flottante s'est peu à peu imposée. Il fallait se mettre d'accord sur des formats de représentation des nombres bien déterminés et spécifier précisément le comportement de chaque opération.

Si $x$ et $y$ sont deux nombres flottants, $x + y$, $x - y$, $xy$, $x/y$, etc. ne sont pas, en général, des nombres flottants. Quelle valeur devait-on renvoyer comme résultat de ces opérations ? Si on note $z$ le résultat exact de l'opération, il semble naturel de renvoyer l'un des deux nombres flottants qui encadrent directement $z$. Lequel des deux renvoyer ? Le plus proche, par exemple. Mais pour certaines applications, il peut être intéressant de renvoyer toujours le plus grand des deux flottants encadrant $z$ : ainsi, on sait que la valeur calculée majore la valeur exacte, ce qui est une information

intéressante. On a donc envie de pouvoir choisir un mode d'arrondi différent, suivant qu'on veut majorer, minorer ou simplement avoir une valeur approchée de la valeur exacte.

En 1985 paraît la norme IEEE-754 : elle définit un cadre rigoureux et contraignant pour le calcul en arithmétique flottante. À partir de sa parution, les systèmes informatiques se mettent peu à peu en conformité avec la norme : tout le monde utilise désormais les mêmes formats de représentation des nombres, et le comportement de chaque opération est bien spécifié et identique sur toutes les machines. La norme définit quatre modes d'arrondi et précise que les opérations doivent être effectuées *avec arrondi correct* : le résultat d'une opération doit être le même que celui qu'on obtiendrait si on calculait la valeur $z$ exactement et qu'on l'arrondissait en utilisant le mode d'arrondi choisi par l'utilisateur. Cette contrainte s'applique aux quatre opérations arithmétiques (addition, soustraction, multiplication, division) ainsi qu'à la racine carrée. En revanche, le comportement des autres fonctions mathématiques (telle que exp, arccos, sinh, etc.) n'est pas spécifié.

La norme apporte un confort qui dépasse le simple cadre de la portabilité du code. Les opérations étant bien spécifiées, on peut borner rigoureusement l'erreur commise par chaque opération et ainsi analyser et contrôler la propagation des erreurs d'arrondi au cours de longs calculs. On peut donc écrire des programmes dont le comportement numérique est certifié. L'utilisation intelligente des modes d'arrondi permet l'émergence de l'arithmétique d'intervalle : une technique qui permet de majorer automatiquement les erreurs d'arrondi. Le respect de la norme permet aussi la conception d'algorithmes dits *exacts* (algorithmes *Fast2Sum* pour l'addition et *de Dekker* pour la multiplication). Si $z$ désigne le résultat exact de $x \diamond y$ (avec $\diamond \in \{+, -, \times\}$), les algorithmes *Fast2Sum* et *de Dekker* permettent de calculer un couple de nombres flottants $(r, \varepsilon)$ tels que $r$ est l'arrondi au plus près de $z$ et $r + \varepsilon = z$. En utilisant habilement ces algorithmes, on peut simuler des formats flottants ayant une plus grande précision.

### Vers une révision de la norme

L'adoption de la norme a été un gros progrès : les fabricants de microprocesseurs s'y sont peu à peu conformés et les programmes sont donc devenus plus précis et plus portables. Mais la norme avait ses limites : elle n'imposait l'arrondi correct que pour les quatre opérations arithmétiques et la racine carrée. Pour les autres fonctions mathématiques, rien n'était spécifié.

Si la norme ne demandait pas l'arrondi correct des fonctions mathématiques telles que l'exponentielle ou les fonctions trigonométriques, c'est qu'on ne savait tout simplement pas fournir l'arrondi correct de ces fonctions. Pour fournir l'arrondi correct d'une fonction numérique $f$, il faut préalablement résoudre un problème connu sous le nom de *dilemme du fabricant de tables*. Supposons qu'on veuille évaluer $f$ avec arrondi correct en $x$. On ne peut généralement pas calculer exactement la valeur $f(x)$ pour la simple raison qu'elle n'est pas représentable. On en calcule donc une valeur approchée $y$. La valeur $y$ est calculée avec une précision plus grande que la précision cible. Mais, même si la précision de $y$ est très grande et que $y$ est très proche de $f(x)$, ils ne s'arrondissent pas forcément à la même valeur. En effet, la fonction d'arrondi est discontinue par nature. Par exemple en arrondi au plus près, la fonction d'arrondi a un point de discontinuité à chaque milieu entre deux nombres flottants consécutifs. Ainsi, même si $y$ et $f(x)$ sont très proches, s'ils sont de part et d'autre de la discontinuité, ils s'arrondissent différemment. Comment, dans ces conditions, pourrait-on garantir l'arrondi correct ?

En 1991, Abraham Ziv — qui travaillait pour IBM — propose une méthode permettant d'obtenir à coup sûr l'arrondi correct. Il s'appuie sur la remarque suivante : si $f(x)$ n'est pas exactement sur un point de discontinuité, il suffit de calculer $y$ suffisamment précisément. En effet, si la distance entre $y$ et $f(x)$ est arbitrairement réduite, ils finissent par être tous les deux du même côté de la discontinuité. La stratégie de Ziv est donc la suivante :

— traiter à part les cas où $f(x)$ est exactement sur un point de discontinuité (des théorèmes de théorie des nombres transcendants permettent de lister exhaustivement ces cas exacts pour

les fonctions telles que exp, log, sin, etc.) ;
— calculer des approximations successives $y_1$, $y_2$, ..., $y_n$ de $f(x)$ de plus en plus précises, jusqu'à pouvoir décider l'arrondi correct.

Ziv fait cependant remarquer que le nombre $n$ d'étapes n'est pas connu à l'avance et pourrait être très grand. De même la précision de l'approximation $y_n$ correspondante pourrait être telle que la mémoire de l'ordinateur ne suffise pas à la représenter. Ziv s'appuie sur cette stratégie pour écrire une bibliothèque de fonctions mathématiques (une libm, dans le jargon) appelée `libultim`. Sa bibliothèque contient un test pour empêcher de faire un trop grand nombre d'étapes ; si l'arrondi correct n'a pas pu être fourni au-delà, le calcul échoue.

Ce premier pas franchi par Ziv est décisif. Cela étant, les industriels restent dubitatifs. Pour eux, l'arrondi correct est un luxe qu'on ne peut pas se payer en pratique. Imaginons qu'un missile ait besoin d'évaluer un cosinus pour corriger sa trajectoire en cours de vol ; est-il raisonnable de se lancer dans des approximations de plus en plus précises pour calculer l'arrondi correct du cosinus ? Le missile aura le temps de manquer sa cible et d'exploser avant que le calcul ne soit fini. Et tout ça, alors qu'une simple valeur approchée aurait parfaitement fait l'affaire ! L'arrondi correct permet la portabilité du code, certes, mais il faut rester réaliste et admettre qu'on ne peut se permettre d'utiliser une bibliothèque dont le temps d'exécution dans le pire des cas est très grand, voire inconnu.

À partir de la fin des années 90, Jean-Michel Muller et ses collègues se saisissent du problème : ils étudient expérimentalement la précision nécessaire pour le calcul de $y_n$ dans le pire des cas. Leurs observations montrent que cette précision est de l'ordre de deux fois la précision du format flottant vers lequel on souhaite arrondir. De plus, comme il n'y a qu'un nombre fini de nombres flottants $x$, il est en principe possible de calculer (par essais exhaustifs) cette précision dans le pire des cas. La thèse de Vincent Lefèvre, publiée en 2000, s'attaque à ce calcul. En utilisant des algorithmes plus fins que la brutale recherche exhaustive, Lefèvre parvient à calculer effectivement la précision nécessaire dans le pire des cas et elle est effectivement de l'ordre de deux à trois fois la précision cible.

Ces travaux permettent de penser qu'il est possible d'écrire une libm dont les fonctions soient correctement arrondies et dont le temps d'exécution dans le pire des cas soit connu et raisonnable. C'est le but que se donne l'équipe Arénaire à travers la réalisation d'un prototype : `CRlibm`. La thèse de David Defour, publiée en 2003, étudie le cas de la fonction exponentielle ; c'est la première fonction implémentée dans `CRlibm`. Par la suite, d'autres fonctions sont ajoutées, le développement de la bibliothèque s'organise, les méthodes s'affinent. À ce jour, `CRlibm` compte une vingtaine de fonctions. Le temps d'exécution moyen de `CRlibm` est le même que le temps d'exécution moyen d'une libm usuelle non correctement arrondie. Le temps d'exécution dans le pire des cas est environ dix fois plus grand que le temps d'exécution moyen.

La norme IEEE-754 a été révisée en 2008. La nouvelle version recommande désormais (il ne s'agit pas toutefois d'une obligation contraignante) que *toutes* les opérations (et pas simplement les opérations arithmétiques) soient correctement arrondies. Des fabricants de microprocesseurs tels qu'Intel ou AMD ont participé au processus de révision et d'adoption de la nouvelle norme. Qu'ils aient accepté cette nouvelle spécification montre que `CRlibm` a atteint son but : démontrer que l'exigence d'arrondi correct n'était pas incompatible avec les contraintes industrielles et pouvait être satisfaite pour un surcoût modique par rapport à une libm classique.

## Le développement d'une fonction en précision fixée

De nombreux problèmes traités dans cette thèse prennent leur source dans le développement de `CRlibm`. Cependant, la portée des résultats de ce manuscrit est plus générale : ils sont utiles dès lors qu'on cherche à écrire du code optimisé pour évaluer une fonction $f$ en précision fixée (que ce

soit avec arrondi correct ou non, la précision pouvant aller de quelques bits seulement à plus d'une centaine de bits). Nous décrivons à présent les grandes lignes du processus d'implémentation d'une fonction en précision fixée.

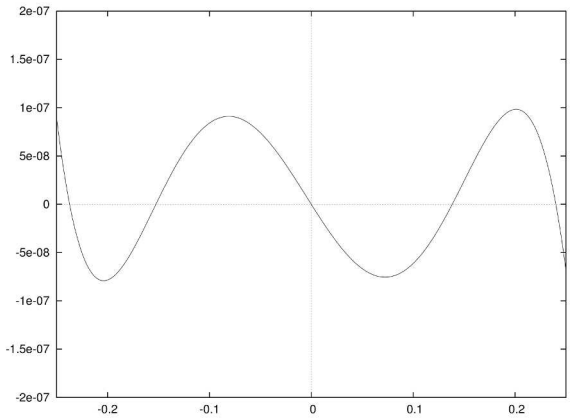
Soit $f$ une fonction numérique univariée définie sur tout ou partie de $\mathbb{R}$. On cherche à implémenter $f$ avec une erreur relative inférieure à une certaine borne $2^{-t}$. La valeur $t$ est la précision avec laquelle on souhaite calculer $f$. Elle est déterminée préalablement : par exemple, on peut prendre $t = 53$ si on vise une implémentation au format double précision (sans arrondi correct) mais on pourrait aussi bien prendre $t = 10$ si on ne souhaite connaître que quelques chiffres significatifs de $f$. Le schéma généralement adopté pour le développement de $f$ est alors le suivant :

1. Se ramener au problème de l'évaluation d'une fonction $g$ sur un petit intervalle $[a, b]$. La fonction $g$ peut être $f$ elle-même ou une fonction auxiliaire à partir de laquelle on peut reconstruire $f$. Pour se ramener à un petit intervalle, on utilise des propriétés algébriques telles que $\exp(2x) = \exp(x)^2$ lorsqu'elles existent. Sinon, lorsque le domaine de définition de $f$ est borné, on peut le couper en plusieurs petits intervalles et travailler sur chacun d'entre eux. Cette étape est essentiellement une affaire de spécialiste et est traitée au cas par cas.
2. Trouver un polynôme $p$ de degré le plus petit possible et à coefficients en virgule flottante qui approche $g$ avec une erreur relative inférieure à $2^{-t-1}$.
3. Écrire le code en langage C permettant d'évaluer $p$ avec la précision requise. Comme chaque opération durant l'évaluation va engendrer une erreur d'arrondi, il faut choisir soigneusement la précision de chaque opération de façon que l'erreur d'évaluation totale reste inférieure à $2^{-t-1}$. Ainsi la somme des deux erreurs (erreur d'approximation entre $p$ et $g$, d'une part, et erreur dans l'évaluation de $p$, d'autre part) reste inférieure à la borne $2^{-t}$.

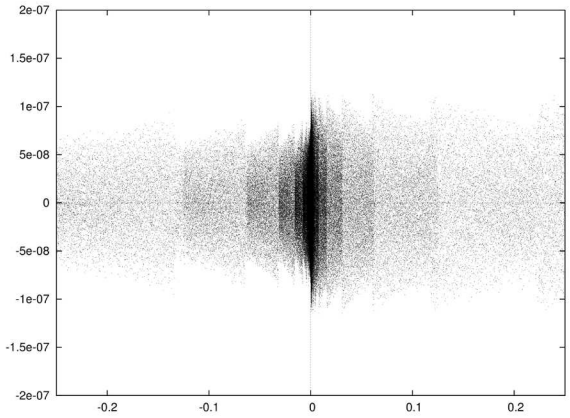
La figure A montre l'erreur d'approximation et l'erreur d'évaluation dans un cas typique. Comme on le voit, les deux erreurs sont de nature très différente : l'erreur d'évaluation a une apparence aléatoire alors que l'erreur d'approximation est très régulière. Pour certifier que l'implémentation finale est correcte, il faut borner rigoureusement l'une et l'autre. Les travaux de thèse de Guillaume Melquiond et Christoph Lauter permettent de traiter l'erreur d'évaluation de façon très satisfaisante. L'erreur d'approximation, malgré son caractère très régulier, n'est pas facile à borner finement de manière rigoureuse (en fait, elle est très facile à estimer numériquement, mais il est difficile d'obtenir un résultat mathématiquement prouvé). Le chapitre 3 du présent manuscrit explique les raisons de cette difficulté et propose un algorithme efficace pour résoudre ce problème.

Nous nous sommes particulièrement intéressés à l'étape 2 du schéma ci-dessus. La première partie de cette étape consiste à trouver un polynôme d'approximation $p^\star$ *à coefficients réels* qui approche la fonction $g$. Dans un deuxième temps, on utilise ce polynôme à coefficients réels pour obtenir un polynôme d'approximation $p$ *à coefficients flottants.* Le chapitre 2 est quasiment intégralement consacré à ces deux sujets. Il est parfois intéressant d'utiliser une fraction rationnelle plutôt qu'un polynôme pour approcher $g$. La recherche de bonnes approximations rationnelles à coefficients flottants est l'objet de la fin du chapitre 2.

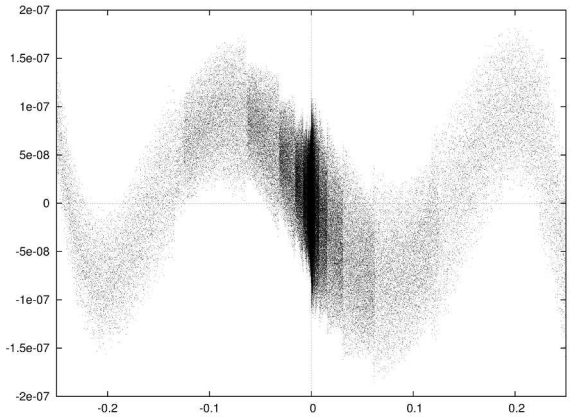
Les algorithmes de Lefèvre, la recherche d'approximations polynomiales de qualité, le calcul certifié du maximum de l'erreur sur un intervalle... tous ces algorithmes nécessitent d'évaluer la fonction $f$ ou $g$ de nombreuses fois avec une précision bien supérieure à la précision visée. Autrement dit, pour pouvoir écrire une implémentation efficace de $f$ en précision fixée, il nous faut avoir au préalable une implémentation efficace de $f$ en précision plus grande. Si on essaie d'appliquer le même schéma pour cette implémentation en grande précision, on arrive à la conclusion qu'il faut d'abord une implémentation de $f$ en précision encore plus grande, etc. Il nous faut donc sortir de ce schéma, et accepter de faire du code moins efficace et plus générique : autrement dit, réaliser une implémentation en précision arbitraire. La précision à laquelle on souhaite connaître $f(x)$ n'est alors plus fixée mais est un paramètre de l'algorithme. L'implémentation de fonctions en précision




(a) Erreur d'approximation $\varepsilon = p/f - 1$




(b) Erreur d'évaluation (due aux arrondis)




(c) Erreur totale

**Figure A :** Graphes des différentes erreurs commises dans l'implémentation d'une fonction

arbitraire est un domaine de recherche en soi ; nous en étudions quelques aspects sur l'exemple des fonctions erf et erfc au chapitre 1.

Pour faciliter le développement de `CRlibm` en automatisant certaines tâches et en permettant aux développeurs de réaliser rapidement des expérimentations numériques, nous avons développé, principalement avec C. Lauter, un logiciel appelé Sollya. Ce logiciel a permis à Lauter d'automatiser quasiment complètement les étapes 2 et 3 du schéma général décrit plus haut, allégeant considérablement la charge de travail (et donc le temps) nécessaire à l'implantation de nouvelles fonctions dans `CRlibm`. Sollya sera donc le support privilégié des exemples proposés tout au long du manuscrit et, à ce titre, il est brièvement présenté entre les chapitres 1 et 2. Sollya a maintenant atteint une certaine maturité et son développement actuel dépasse le strict cadre original d'outil d'aide aux développeurs de `CRlibm`. Il pourrait, à terme, intéresser toute personne souhaitant faire des calculs numériques dans un environnement rapide et fiable. Le chapitre 4 est l'occasion de revenir sur les points forts et les points faibles de Sollya et d'envisager les perspectives pour son avenir.

# Contributions de la thèse

Cette thèse aborde principalement trois sujets différents, tous liés au problème de l'évaluation efficace de fonctions numériques d'une variable réelle :

— le premier sujet est celui de l'évaluation d'une fonction en précision arbitraire, avec des bornes d'erreur effectives. Il fait l'objet du chapitre 1 ;

— le deuxième sujet concerne l'approximation de fonctions par des polynômes dont les coefficients vérifient un certain nombre de contraintes. Ce problème constitue le cœur de la thèse et fait l'objet du chapitre 2 ;

— le chapitre 3 aborde le dernier sujet, à savoir le calcul de la norme sup d'erreurs d'approximation, par des méthodes certifiées.

En toile de fond de tout le manuscrit, on retrouve le logiciel Sollya, développé pendant la thèse, sorte de boîte à outils dont l'ambition est d'offrir un environnement sûr et efficace pour faciliter le travail de ceux qui développent des codes numériques.

### Évaluation en précision arbitraire

Le chapitre 1 s'intéresse à l'implémentation d'une fonction en précision arbitraire. Dans notre implémentation, les bornes d'erreurs sont effectives et ne sont pas de simples estimations comme c'est souvent le cas. Les méthodes mises en œuvre ne sont pas nouvelles ; ce qui fait l'originalité de ce travail, c'est que la démarche générale et les preuves sont bien détaillées afin de servir de référence et de modèle pour l'implantation d'autres fonctions. Ce travail a fait l'objet de deux articles :

> [CR08] S. Chevillard et N. Revol, *Computation of the error function erf in arbitrary precision with correct rounding*. Dans J. D. Bruguera et M. Daumas (éditeurs) : *RNC 8 Proceedings, 8th Conference on Real Numbers and Computers*, pages 27–36, 2008.

> [Che09] S. Chevillard, *The functions* erf *and* erfc *computed with arbitrary precision*. Rapport de recherche RR2009-04, laboratoire de l'informatique du parallélisme (LIP), 46, allée d'Italie, 69 364 Lyon Cedex 07, 2009. Disponible à l'adresse `https://hal.archives-ouvertes.fr/ensl-00356709`.

L'article [Che09] est beaucoup plus complet et détaillé que [CR08]. Il est actuellement soumis pour publication.

### Approximation sous contraintes

Le chapitre 2 se compose principalement de trois parties. La première partie (section 2.2) est constituée de rappels concernant l'approximation par des polynômes à coefficients réels. Certains résultats de ce domaine sont peu connus : nous avons donc essayé d'en offrir une vision synthétique et unifiée, illustrée de nombreux exemples.

La deuxième partie est la section 2.3, qui présente nos recherches sur l'approximation polynomiale à coefficients contraints. La plupart des résultats décrits sont nouveaux. Plus précisément, les sections 2.3.2 et 2.3.3 présentent des résultats dûs à Brisebarre, Muller, Tisserand et Torres (voir [BMT06]). Durant la thèse, nous avons amélioré ces méthodes et essayé d'en systématiser l'usage. En section 2.3.4 et suivantes, nous présentons un nouvel algorithme (l'algorithme `fpminimax`). Les idées importantes se trouvaient déjà dans mon mémoire de master et dans l'article de conférence qui a suivi :

> [Che06] S. CHEVILLARD, *Polynômes de meilleure approximation à coefficients flottants*. Mémoire de master, École normale supérieure de Lyon, 46, allée d'Italie, 69 364 Lyon Cedex 07, 2006. Disponible à l'adresse `http://www.ens-lyon.fr/LIP/Pub/Rapports/DEA/DEA2006/DEA2006-03.ps.gz`.

> [BC07] N. BRISEBARRE et S. CHEVILLARD, *Efficient polynomial $L^\infty$-approximations*. Dans P. KORNERUP et J. M. MULLER (éditeurs) : *18th IEEE Symposium on Computer Arithmetic*, pages 169–176, Los Alamitos, CA, 2007. IEEE Computer Society.

Nos travaux sur ce sujet, tels qu'ils sont présentés dans le présent manuscrit, sont bien plus aboutis qu'ils ne l'étaient au moment de ces publications. En particulier, nous développons une vision synthétique qui montre que l'algorithme `fpminimax` et l'algorithme de Brisebarre et co-auteurs sont complémentaires l'un de l'autre. Ceci devrait faire l'objet d'une prochaine publication.

La troisième partie de ce chapitre (section 2.4) s'intéresse à l'approximation de fonctions par des fractions rationnelles. Un algorithme dû à Ercegovac (appelé *E-méthode*) permet d'évaluer efficacement certaines fractions rationnelles. Pour qu'une fraction rationnelle puisse être évaluée par la E-méthode, il faut que ses coefficients vérifient certaines contraintes. Nous présentons un algorithme pour trouver de très bonnes approximations rationnelles vérifiant ces contraintes. Ce travail a fait l'objet d'une publication :

> [BCE+08] N. BRISEBARRE, S. CHEVILLARD, M. D. ERCEGOVAC, J. M. MULLER et S. TORRES, *An Efficient Method for Evaluating Polynomial and Rational Function Approximations*. Dans *ASAP 08, Conference Proceedings, IEEE 19th International Conference on Application-Specific Systems, Architectures and Processors*, pages 245–250, Los Alamitos, CA, 2008. IEEE Computer Society.

L'algorithme que nous avons développé est encore très imparfait ; nous envisageons de l'améliorer et de l'automatiser. À terme, nous devrions pouvoir synthétiser automatiquement un circuit pour évaluer une fonction en utilisant la E-méthode. Une telle synthèse automatique permettra de comparer les performances pratiques de la E-méthode par rapport à des méthodes plus conventionnelles (approximation polynomiale par exemple).

### Calcul certifié de normes sup

Le chapitre 3 aborde la question du calcul automatique et certifié de la norme sup de fonctions. Ce problème se pose naturellement lorsqu'on implémente une fonction $f$ en précision fixée : en général, $f$ est remplacée par une approximation $p$, et il faut donc majorer de façon fine l'erreur entre $p$ et $f$. Ceci revient à calculer la norme sup de la fonction d'erreur $p - f$ ou $p/f - 1$.

Nous rappelons d'abord les algorithmes classiques d'optimisation globale certifiée (en particulier l'algorithme de Hansen) et nous montrons en quoi la nature particulière des fonctions $p-f$ et $p/f-1$ rend ces méthodes inefficaces. C'est ce qui nous a amené à proposer des algorithmes pour traiter spécifiquement ce problème. Nos contributions à ce sujet ont fait l'objet de deux articles distincts :

[CL07] S. Chevillard et Ch. Lauter, *A certified infinite norm for the implementation of elementary functions*. Dans A. Mathur, W. E. Wong et M. F. Lau (éditeurs) : *QSIC 2007, Proceedings of the Seventh International Conference on Quality Software*, pages 153–160, Los Alamitos, CA, 2007. IEEE Computer Society. Une version longue est disponible sous forme du rapport de recherche RR2007-26, laboratoire de l'informatique du parallélisme, à l'adresse `https://hal.archives-ouvertes.fr/ensl-00119810/`.

[CJL09] S. Chevillard, M. Joldeş et Ch. Lauter, *Certified and fast computation of supremum norms of approximation errors*. Dans J. D. Bruguera, M. Cornea, D. DasSarma et J. Harrison (éditeurs) : *19th IEEE SYMPOSIUM on Computer Arithmetic*, pages 169–176, Los Alamitos, CA, 2009. IEEE Computer Society.

Dans [CL07], nous avons proposé un algorithme de type *algorithme de Hansen* pour calculer la norme sup d'une fonction de façon certifiée. L'originalité de ce travail repose en fait sur l'algorithme que nous utilisons pour évaluer une expression par arithmétique d'intervalle. Cet algorithme correspond à la commande `evaluate` de Sollya et il permet de réduire les phénomènes de décorrélation dans l'évaluation des fonctions d'erreur $p - f$ et $p/f - 1$. En outre, nous expliquons comment traiter les fausses singularités qui peuvent apparaître dans l'expression $p/f - 1$ : il s'agit d'une application de la règle de L'Hôpital dans le contexte de l'arithmétique d'intervalle.

Ce premier algorithme était très imparfait et ne permettait pas de traiter toutes les normes sup dont nous avions besoin en pratique. Le deuxième algorithme, que nous détaillons en section 3.3 et suivantes, est beaucoup plus satisfaisant. Il est complètement décrit (mais plus brièvement que dans le présent manuscrit) dans l'article correspondant [CJL09].

### Le logiciel Sollya

Le développement de Sollya a représenté une part importante du travail de thèse. Nous en décrivons les grandes lignes dans un chapitre non numéroté (page 67 et suivantes). Le chapitre 4, en fin de document, analyse les forces et les faiblesses du logiciel et évoque quelques pistes pour son développement futur.

Il n'est pas aisé de quantifier la contribution scientifique d'un logiciel. Quelques chiffres peuvent donner une idée : Sollya est constitué d'environ 65 000 lignes de code, écrites principalement avec Christoph Lauter. Le logiciel est fourni avec un manuel d'utilisation de 120 pages environ. Pour ce que nous en savons, il compte actuellement une vingtaine d'utilisateurs (des chercheurs académiques, mais aussi des ingénieurs). Nous espérons faire la démonstration de son utilité en filigrane des nombreux exemples qui parsèment ce manuscrit.

Le manuel utilisateur constitue une bonne référence introductive. Il comporte une présentation de la philosophie générale de l'outil ; en outre, l'ensemble des commandes disponibles est documenté, ce qui permet de se rendre compte des possibilités offertes par Sollya :

[CJJL] S. Chevillard, M. Joldeş, N. Jourdan et Ch. Lauter, *User's Manual of the Sollya Tool*. Disponible à l'adresse `http://sollya.gforge.inria.fr/`.

Chapitre 1

# Évaluation en précision arbitraire : l'exemple de la fonction erf

*Ce chapitre présente la problématique de l'évaluation d'une fonction en précision arbitraire à travers l'exemple de la fonction d'erreur* erf. *Notre but est ici de montrer les spécificités propres à l'implantation d'une fonction en précision arbitraire. L'originalité de ce travail ne réside ni dans le choix de la fonction (il existe déjà des implantations de la fonction* erf *en précision arbitraire), ni dans les méthodes utilisées : les formules d'approximations sont bien connues, le schéma d'évaluation utilisé a été proposé par Smith en 1989, l'analyse d'erreur utilise des techniques classiques. Nos contributions sont de trois ordres :*

- *l'implantation proposée garantit une borne d'erreur effective : en notant $t'$ la précision cible demandée par l'utilisateur, notre implantation garantit que l'erreur relative entre la valeur calculée $y$ et la valeur exacte* $\mathrm{erf}(x)$ *est inférieure à $2^{-t'}$. La plupart des implantations disponibles se contentent d'assurer que cette erreur reste $\mathcal{O}(2^{-t'})$, mais sans fournir de majoration effective de l'erreur ;*
- *nous nous comparons à la bibliothèque `MPFR` qui est une référence en matière d'implantation en précision arbitraire et qui garantit également les bornes d'erreur. Le soin apporté à notre implantation nous permet d'être aussi rapide que `MPFR` en faible précision et d'être nettement plus rapide que `MPFR` lorsque la précision augmente ;*
- *enfin, nous expliquons précisément les algorithmes utilisés et décrivons en détail toutes les preuves. Ainsi, à travers l'exemple d'une fonction, nous fournissons un modèle qui pourra être repris pour l'implantation d'autres fonctions.*

## 1.1 Introduction

Dans ce premier chapitre, nous allons nous intéresser, sur un exemple, au problème de l'évaluation d'une fonction en précision arbitraire. La problématique de l'évaluation en précision arbitraire est la suivante : étant donnée une fonction numérique $f$, on cherche à en calculer une valeur approchée en un point $x$, avec une erreur bornée par une certaine quantité $\varepsilon$. L'algorithme a donc deux paramètres : le point $x$ et la borne $\varepsilon$. On parle de précision arbitraire, puisque $\varepsilon$ peut être choisi arbitrairement petit par l'utilisateur.

Au chapitre 2, nous nous intéresserons au problème de l'évaluation en précision fixée : cette fois, $\varepsilon$ ne sera plus un paramètre de l'algorithme, mais une donnée connue d'avance, au moment de la conception de l'algorithme. En travaillant soigneusement, on peut alors écrire un algorithme spécifiquement adapté à la précision demandée et donc plus efficace que l'algorithme de précision

arbitraire. Nous étudierons quelques-uns des outils qui permettent d'écrire un algorithme spécifique efficace en précision fixée.

Pour fonctionner, ces outils ont besoin d'évaluer la fonction $f$ avec une précision plus grande que la précision cible ; par exemple, pour produire une implantation efficace de $f$ avec une précision de 53 bits, les outils pourront avoir besoin d'évaluer $f$ avec une précision allant de 60 à 150 bits. Autrement dit, pour pouvoir produire une implémentation efficace en précision fixée, il faut préalablement pouvoir évaluer la fonction en précision arbitraire. C'est la raison pour laquelle nous étudions le problème de l'approximation en précision arbitraire en premier.

Nous traitons ce problème à travers l'exemple des fonctions erf et erfc. Il illustre bien les questions qu'on est amené à résoudre lors de l'implantation d'une fonction en précision arbitraire : choix d'une formule d'approximation, détermination du rang de troncature, détermination de la précision intermédiaire nécessaire, analyse rigoureuse des erreurs d'arrondi. Évidemment, cette démarche a ses limites et nous ne pourrons aborder, sur un seul exemple, toutes les techniques usuelles de ce domaine : utilisation de fractions continues, de l'itération de Newton, de la moyenne arithmético-géométrique, scindage binaire, etc. Pour une approche plus exhaustive, on pourra se reporter à l'un des nombreux articles sur le sujet, par exemple [Bre80] (ou encore [Bre76, Bre78, FHL+07, Jea00]).

### 1.1.1 Représentation des nombres

Pour évaluer une fonction numérique, il faut travailler de manière approchée avec les nombres réels. Puisque l'ensemble des nombres réels est indénombrable il n'est pas concevable de représenter tous les nombres réels de manière exacte. On doit se limiter à un ensemble dénombrable — voire, le plus souvent, fini — qui servira d'approximation pour représenter l'ensemble de tous les nombres réels.

Plusieurs systèmes de représentations sont concevables [Mul89] : on peut par exemple décider d'approcher l'ensemble des réels :

— par l'ensemble des rationnels (vus comme couples de nombres entiers) ;
— par le début du développement en fraction continue ;
— par le début de l'écriture binaire des nombres ;
— etc.

L'idée de représenter les nombres par le début de leur développement binaire donne lieu à deux formats couramment utilisés : la représentation en *virgule flottante* et la représentation en *virgule fixe*.

#### Virgule flottante

Dans la représentation en virgule flottante, on choisit une base entière $\beta \geq 2$ et un entier $t$ qui est la précision du format. La précision indique combien de chiffres significatifs (en base $\beta$) seront utilisés pour représenter les nombres. L'ensemble des nombres flottants de précision $t$ en base $\beta$ est défini par

$$\mathcal{F}_t = \{0\} \cup \left\{ \frac{m}{\beta^t}\, \beta^e,\ e \in \mathbb{Z},\ |m| \in [\![\beta^{t-1},\ \beta^t - 1]\!] \right\} \quad \text{où} \quad [\![\beta^{t-1},\ \beta^t - 1]\!] = [\beta^{t-1},\ \beta^t - 1] \cap \mathbb{Z}.$$

L'entier $m$ est appelé *la mantisse* et s'écrit sur $t$ chiffres exactement en base $\beta$. L'entier $e$ est *l'exposant*. Il indique l'ordre de grandeur du nombre considéré.

Une autre façon de voir les choses consiste à écrire le développement de $m$ en base $\beta$ : $m = m_1 \cdots m_t$. Les bornes sur $m$ impliquent que $t$ chiffres sont nécessaires et suffisants pour écrire $m$ en base $\beta$ : donc $m_1 \neq 0$. Le nombre flottant de mantisse $m$ et d'exposant $e$ est alors $0, m_1 \cdots m_t \cdot \beta^e$. C'est de là que vient le terme de « virgule flottante » : on ne conserve que les $t$ chiffres les plus

significatifs du nombre, son ordre de grandeur étant donné par l'exposant $e$ (qui indique donc la position de la virgule dans l'écriture usuelle).

Dans la pratique, l'exposant est souvent stocké sur un nombre fini de bits défini à l'avance. L'exposant doit donc rester entre deux bornes $e_{\min}$ et $e_{\max}$. La base $\beta$ la plus répandue est la base 2 ; cependant certains systèmes utilisent d'autres bases (par exemple, Maple ainsi que la plupart des calculettes de poche utilisent la base 10). L'ordinateur russe SETUN, construit en 1958, à l'université de Moscou utilisait la base 3.

La définition d'arithmétique en virgule flottante est régie par une norme qui date de 1985 : la norme IEEE-754 [AI85] (la norme a été récemment révisée [AI08]). La norme définit cinq formats appelés binary32, binary64, binary128, decimal64 et decimal128. Comme leurs noms le laissent penser, les trois premiers sont en base 2 et les deux derniers en base 10. Le format binary32 est aussi connu sous le nom de *simple précision* et le format binary64 sous le nom de *double précision* ou encore format `double`. Le tableau 1.1 rappelle les caractéristiques de ces formats.

| Format | $\beta$ | $t$ | $e_{\min}$ | $e_{\max}$ |
|---|---|---|---|---|
| binary32 | 2 | 24 | −125 | 128 |
| binary64 | 2 | 53 | −1 021 | 1 024 |
| binary128 | 2 | 113 | −16 381 | 16 384 |
| decimal64 | 10 | 16 | −382 | 385 |
| decimal128 | 10 | 34 | −6 142 | 6 145 |

**Figure 1.1 :** Caractéristiques des formats définis dans la norme IEEE-754

La norme définit aussi des valeurs spéciales (NaN, Inf) et précise comment il faut traiter les cas spéciaux tels qu'une division par 0, la racine carrée d'un nombre négatif, etc. En outre, elle explique ce qu'il faut faire lors d'un dépassement de capacité, c'est-à-dire lorsque le résultat d'un calcul est un nombre dont l'exposant n'est pas compris entre $e_{\min}$ et $e_{\max}$. Sauf mention contraire, nous ne nous préoccuperons pas de ces spécificités dans la présente thèse.

La norme ne se contente pas de standardiser la représentation des nombres. Elle explique aussi comment il faut calculer avec. Ainsi, elle définit quatre modes d'arrondi.

**Définition 1.1.** Soit $x$ un nombre réel. Les quatre modes d'arrondi de la norme sont définis de la façon suivante :

- l'arrondi vers le haut de $x$ (arrondi par excès) est le plus petit nombre flottant supérieur ou égal à $x$. On le notera $\Delta(x)$ ;
- l'arrondi vers le bas de $x$ (arrondi par défaut) est le plus grand nombre flottant inférieur ou égal à $x$. On le notera $\nabla(x)$ ;
- l'arrondi vers 0 (troncature) consiste à faire un arrondi par défaut si $x$ est positif et un arrondi par excès si $x$ est négatif ;
- l'arrondi au plus près de $x$ est le nombre flottant $\widehat{x}$ qui minimise $|x - \widehat{x}|$. Au cas où il existe deux nombres flottants atteignant ce minimum, on choisit celui dont la mantisse est paire (on parle d'arrondi au plus près pair).

Si $x$ est un nombre réel, on dit que $\widehat{x}$ est un *arrondi fidèle* de $x$ si $\widehat{x} \in \{\nabla(x), \Delta(x)\}$. Autrement dit, si $x$ est un nombre flottant, $\widehat{x} = x$ et sinon, $\widehat{x}$ est l'un des deux nombres flottants encadrant $x$. La notion d'arrondi fidèle n'est pas définie dans la norme, mais elle nous sera très utile.

En plus de définir des modes d'arrondi, la norme impose que chaque opération arithmétique (addition, soustraction, multiplication, division et racine carrée) soit effectuée avec arrondi correct : le nombre flottant $z$ renvoyé comme résultat de l'opération doit correspondre à l'arrondi (suivant le mode d'arrondi demandé par l'utilisateur) du résultat exact de l'opération. Par exemple, si le mode

d'arrondi choisi par l'utilisateur est l'arrondi par excès, et que l'utilisateur demande à calculer la racine carrée d'un nombre $x$, le résultat renvoyé par l'algorithme doit être $\Delta(\sqrt{x})$, comme si le calcul avait été effectué de façon exacte, puis arrondi. La récente révision de la norme recommande que ce principe de l'arrondi correct soit étendu à toutes les fonctions mathématiques proposées à l'utilisateur (exp, cos, etc.).

### Précision arbitraire

Le cadre défini par la norme est très commode : il donne des règles claires et partagées par tous pour le calcul numérique sur ordinateur. Il autorise la réalisation de codes portables : puisque les règles du jeu sont parfaitement définies, un même programme s'exécutera de la même façon sur deux machines différentes si celles-ci respectent la norme. De plus, puisque chaque opération est effectuée avec arrondi correct, on peut étudier et borner rigoureusement la façon dont se propagent les erreurs d'arrondi au cours d'une longue série de calculs. La norme permet donc la réalisation d'algorithmes dont la qualité numérique est certifiée.

Cependant, la norme ne propose que cinq formats. Comment faire si l'on souhaite travailler avec plus de précision ? En particulier, si on a besoin de faire du calcul numérique avec plusieurs centaines, voire plusieurs milliers de chiffres ? La récente révision de la norme suggère (mais n'impose pas) l'adoption de formats dits *étendus* permettant à l'utilisateur de travailler avec une précision multiple de 32 quelconque (fixée à la compilation). De même, elle évoque les formats dits *extensibles* permettant de faire varier arbitrairement la précision (cependant, là encore, la précision doit être un multiple de 32). Cela étant, ces recommandations ne figuraient pas dans la première version de la norme et ne sont pas encore implémentées en pratique. L'idée d'étendre les définitions et les recommandations de la norme à des formats flottants de précision quelconque était tout de même naturelle ; la bibliothèque `MPFR` [FHL+07] a justement été développée depuis l'année 2000 pour proposer une telle arithmétique flottante de précision arbitraire. Avec `MPFR`, l'utilisateur peut choisir au bit près la précision de chaque nombre. La bibliothèque s'inspire de la norme IEEE-754 et en étend les principes : les quatre modes d'arrondi de la norme sont proposés et les opérations sont toutes correctement arrondies. À ce jour, `MPFR` fournit une implantation des opérations arithmétiques (addition, soustraction, multiplication, division, racine carrée), une cinquantaine de fonctions élémentaires et spéciales (exponentielle, fonctions de Bessel, etc.), ainsi que des routines de conversions et d'entrées/sorties.

La norme IEEE-754 et la bibliothèque `MPFR` n'adoptent pas exactement la même convention de normalisation pour la mantisse des nombres flottants.[1] Pour `MPFR`, la mantisse est un nombre compris entre 1/2 et 1 alors que dans la norme, la mantisse est un nombre compris entre 1 et 2. Ceci induit un décalage de 1 dans le définition de l'exposant : le nombre $1, m_2 \cdots m_t$ (dont l'exposant est 0 pour la norme) est vu comme le nombre $0, 1m_2 \cdots m_t \cdot 2^1$ par `MPFR` (qui considère donc que son exposant est 1). Dans tout le présent document, nous adoptons la même convention que `MPFR`.

### Virgule fixe

La virgule flottante est sans doute le format le plus utilisé pour les calculs numériques logiciels. Un autre format classique existe, généralement plutôt réservé à des applications spécifiques et proches du matériel : il s'agit de la représentation en virgule fixe.

La représentation en virgule fixe dépend, elle aussi, d'une base $\beta$ et d'une précision $t$. Par ailleurs, on fixe une fois pour toute un entier relatif $E$ qui va servir à indiquer la position de la virgule. Ceci définit un format de virgule fixe. Un nombre dans ce format est la donnée d'un entier

1. Il semble que ce choix historique — surprenant de la part des développeurs de `MPFR` — vienne du fait que le bit de poids fort n'est pas implicite dans `MPFR`. On peut estimer qu'il est alors plus naturel d'avoir une mantisse dans l'intervalle [1/2, 1[.

$m$ s'écrivant sur au plus $t$ bits ; autrement dit $|m| \in [\![0,\, \beta^t - 1]\!]$. Le nombre représenté par $m$ est

$$a = \frac{m}{\beta^E}.$$

Si $E$ est positif, par exemple, tous les nombres seront donc représentés avec au plus $E$ chiffres après la virgule. D'où le nom de *virgule fixe*. L'arithmétique en virgule fixe se ramène essentiellement à de l'arithmétique entière, ce qui en rend l'implantation simple et efficace. Son inconvénient est que l'ordre de grandeur des nombres à manipuler est choisi d'avance. Cela nécessite donc en général une analyse préalable des ordres de grandeurs des nombres impliqués dans un calcul, de façon à choisir $E$ correctement.

## 1.2 Cadre général

Nous abordons maintenant le cœur du sujet de ce chapitre, à savoir l'évaluation des fonctions erf et erfc en précision arbitraire. La fonction erf, appelée fonction d'erreur, est définie de la façon suivante :

$$\mathrm{erf} : x \mapsto \frac{2}{\sqrt{\pi}} \int_0^x e^{-v^2}\, \mathrm{d}v.$$

Elle est parfois aussi appelée *intégrale de probabilité*, auquel cas erf représente l'intégrale seule, sans le facteur de normalisation $2/\sqrt{\pi}$ [Leb65]. La fonction d'erreur complémentaire, notée erfc, est définie par $\mathrm{erfc} = 1 - \mathrm{erf}$. Ces deux fonctions sont bien définies et analytiques sur tout le plan complexe. Néanmoins, nous nous limiterons dans ce chapitre à leur étude et leur évaluation sur la droite réelle.

Ces fonctions sont importantes car on les rencontre dans de nombreuses branches des mathématiques appliquées, en particulier en théorie des probabilités. En effet, si $X$ est une variable aléatoire suivant une distribution gaussienne de moyenne 0 et d'écart-type $1/\sqrt{2}$, la probabilité $P(-x \leq X \leq x)$ est égale à $\mathrm{erf}(x)$ (figure 1.2). On pourra consulter [Leb65], par exemple, pour d'autres applications des fonctions d'erreur.

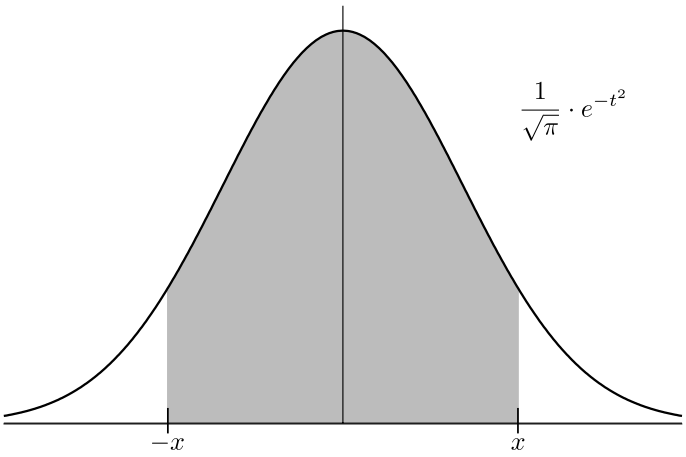



**Figure 1.2 :** La valeur $\mathrm{erf}(x)$ est la probabilité qu'une certaine variable aléatoire gaussienne se situe dans l'intervalle $[-x,\, x]$.

Nous allons étudier comment implémenter les fonctions erf et erfc en arithmétique flottante et avec une précision arbitraire. Une telle implémentation en précision arbitraire se révèle nécessaire dans un certain nombre de cas comme :

— lorsqu'on désire calculer des valeurs approchées de fonctions avec une grande précision ;

— lorsqu'on souhaite vérifier les qualités numériques de bibliothèques implémentant les mêmes fonctions, mais avec une précision plus faible ;
— lorsqu'on souhaite calculer des polynômes d'approximation de fonctions fournissant une précision donnée. Nous aurons l'occasion de revenir sur ce point dans le chapitre 2.

Un bon aperçu des multiples applications de la précision arbitraire est fournie par l'introduction de [Bre80].

Nous allons approcher les nombres réels en utilisant la bibliothèque `MPFR`, donc en utilisant des nombres flottants en base 2 de précision arbitraire. `MPFR` autorise une plage d'exposants très large (typiquement $e \in [\![-2^{32}, 2^{32}]\!]$) et on négligera donc les dépassements de capacité.

Notre but est le suivant : étant donnés $t$ et $t'$ dans $\mathbb{N}^\star$ ainsi que $x \in \mathcal{F}_t$, calculer une valeur $y$ approchant $\mathrm{erf}(x)$ avec une erreur relative inférieure à $2^{-t'}$. Plus formellement, nous voulons

$$\exists \delta \in \mathbb{R},\ \mathrm{erf}(x) = y(1+\delta) \quad \text{où} \quad |\delta| \leq 2^{-t'}.$$

Plusieurs bibliothèques et logiciels implémentent déjà l'arithmétique flottante en précision arbitraire. Outre la bibliothèque `MPFR`, on peut citer par exemple la bibliothèque historique de Brent `MP` [Bre78] écrite en Fortran, la bibliothèque C++ `Arprec` de Bailey [BHLT] et les logiciels très connus Mathematica et Maple. `MPFR` fournit l'arrondi correct, donc la valeur $y$ retournée est l'arrondi de la valeur exacte $\mathrm{erf}(x)$ à la précision cible $t'$. Les autres bibliothèques et logiciels ne garantissent pas en général une propriété aussi forte : au mieux, ils promettent simplement que l'erreur relative entre $y$ et la valeur exacte est $\mathcal{O}(2^{-t'})$.

Cela étant, `MPFR` pourrait ne pas s'arrêter sur certaines entrées. En effet, `MPFR` utilise généralement la stratégie de Ziv [Ziv91] qui peut être brièvement résumée comme suit :

1. On choisit une précision intermédiaire $t_1 \geq t'$. On calcule une valeur approchée $y_1$ de $f(x)$ en précision $t_1$ ainsi qu'une borne $\varepsilon_1$ sur l'erreur absolue :

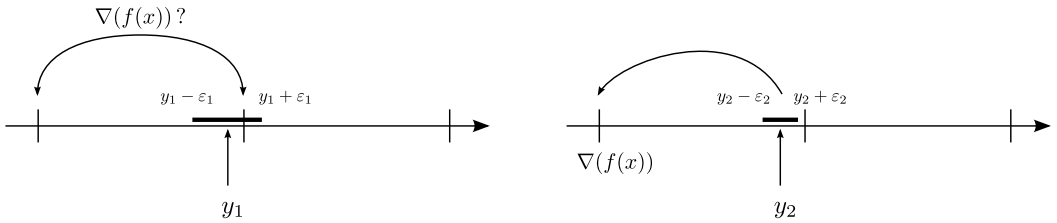
   $$y_1 \in [f(x) - \varepsilon_1,\ f(x) + \varepsilon_1].$$

   Si $f(x) - \varepsilon_1$ et $f(x) + \varepsilon_1$ s'arrondissent à la même valeur en précision $t'$, alors c'est aussi le cas de tous les autres points de l'intervalle. Cette valeur est donc bien l'arrondi correct de $f(x)$.
2. Mais si $f(x) - \varepsilon_1$ et $f(x) + \varepsilon_1$ s'arrondissent à deux valeurs différentes, on ne peut pas savoir quelle valeur est l'arrondi correct. On choisit donc une nouvelle précision $t_2 > t_1$ et on calcule une nouvelle valeur approchée $y_2$ ainsi qu'une borne $\varepsilon_2$. On refait le test, et ainsi de suite.

Supposons par exemple qu'on travaille en arrondi dirigé (arrondi par défaut ou arrondi par excès) et que $f(x)$ soit exactement représentable sur $t'$ bits. Dans ce cas, quel que soit $\varepsilon > 0$, les nombres $f(x) - \varepsilon$ et $f(x) + \varepsilon$ s'arrondissent différemment. La stratégie de Ziv ne s'arrêtera donc jamais sur un tel cas. À l'inverse, si $f(x)$ n'est pas exactement représentable sur $t'$ bits, on peut trouver une valeur de $\varepsilon$ suffisamment petite pour que $f(x) - \varepsilon$ et $f(x) + \varepsilon$ s'arrondissent à la même valeur et la stratégie finira donc par s'arrêter. Ce phénomène est connu sous le nom de *dilemme du fabricant de tables* (Table Maker's Dilemma ou encore TMD en anglais) et est illustré en figure 1.3.

Pour les fonctions telles que exp, sin, ln, etc., la liste des cas exacts est connue (ce sont les identités bien connues $\exp(0) = 1$, $\sin(0) = 0$, $\ln(1) = 0$, etc.). On est assuré qu'il n'y a pas d'autres cas exacts par le théorème de Lindemann-Weierstrass [Bak75, Chap. 1]. On peut donc les traiter séparément. La stratégie de Ziv n'est alors utilisée que si $f(x)$ n'est pas un cas exact.

En ce qui concerne erf et erfc, 0 est un cas exact évident mais rien ne garantit qu'il n'y a pas d'autres cas exacts (même si on estime que c'est peu probable). Sur ces hypothétiques cas exacts, `MPFR` bouclerait indéfiniment. C'est pourquoi nous ne cherchons pas à fournir l'arrondi correct mais simplement à garantir que l'erreur relative finale soit inférieure à $2^{-t'}$. Il s'agit d'une condition plus




(a) L'arrondi ne peut pas être décidé (b) L'arrondi peut être décidé

**Figure 1.3 :** Illustration du dilemme du fabricant de tables, en arrondi par défaut

forte que de demander simplement que l'erreur soit un $\mathcal{O}(2^{-t'})$ puisque nous demandons que la borne d'erreur soit effective. Mais contrairement à `MPFR`, nous avons la garantie que notre implémentation s'arrête sur toutes les entrées. Notons au passage, qu'il ne serait pas difficile de mettre en œuvre la stratégie de Ziv en utilisant notre implémentation de façon à fournir l'arrondi correct dans tous les cas où c'est possible.

Le travail présenté dans ce chapitre n'est pas nouveau quant à la nature des techniques utilisées : nous utilisons des formules d'approximation classiques de erf et erfc, implémentons une technique de sommation proposée par Smith en 1989 [Smi89] et analysons les erreurs d'arrondi à l'aide de techniques classiques. En revanche, nous faisons une étude rigoureuse des erreurs qui se propagent de façon que la précision de travail soit choisie au plus juste tout en garantissant la correction de l'algorithme. Nous appliquons cette démarche à trois méthodes d'approximation différentes. Nous comparons leur efficacité en fonction du point d'évaluation $x$ et de la précision cible $t'$, de façon à obtenir finalement l'implémentation la plus efficace possible. Les techniques utilisées dans ce chapitre donnent un aperçu assez précis de ce que peut être le schéma général de développement d'une fonction en précision arbitraire, schéma qui pourra être suivi pour l'implémentation d'autres fonctions.

Dans un premier temps, la section 1.3 décrit des propriétés générales de erf et erfc et définit les trois formules d'approximation qui permettront l'implémentation en précision arbitraire. Ensuite, dans la section 1.4, nous rappelons des techniques classiques permettant d'analyser les erreurs d'arrondi qui se propagent inévitablement dans les calculs. Nous décrivons alors complètement, en section 1.5, les trois algorithmes. En particulier, nous donnons des bornes rigoureuses et explicites pour le rang de troncature utilisé et les erreurs d'arrondi. Notre implémentation est écrite en C et utilise la bibliothèque `MPFR`. Nous sommes actuellement en discussion avec les développeurs de `MPFR` pour l'y intégrer. Finalement, en section 1.6, nous étudions expérimentalement lequel des trois algorithmes est le plus rapide en fonction du point d'approximation $x$ et de la précision cible $t'$.

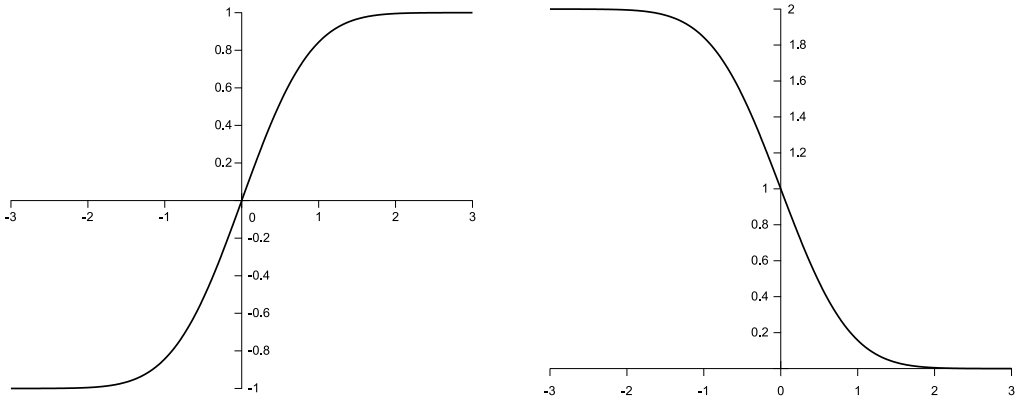
## 1.3 Aperçu général des algorithmes

On voit trivialement que erf est impaire ; on peut donc se limiter au calcul de $\mathrm{erf}(x)$ lorsque $x > 0$ sans perte de généralité. Sauf mention explicite contraire, on supposera donc toujours que $x$ est strictement positif.

La fonction erf tend vers 1 en $+\infty$. Aussi, pour de grandes valeurs de $x$, la représentation binaire de $\mathrm{erf}(x)$ est de la forme

$$0,\underbrace{11\cdots11}_{\substack{\text{beaucoup}\\ \text{de '1'}}} b_1 b_2 b_3 \cdots$$

Il est donc plus intéressant, pour de telles valeurs de $x$, d'évaluer la fonction complémentaire $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$. Les graphes des deux fonctions sont représentés en figure 1.4.




(a) Graphe de erf (b) Graphe de erfc

**Figure 1.4 :** Graphes des fonctions erf et erfc

Parmi les formules données dans [AS65], nous retenons les trois suivantes qui sont appropriées pour l'évaluation en précision arbitraire (équations 7.1.5, 7.1.6, 7.1.23 et 7.1.24 de [AS65]) :

$$\operatorname{erf}(x) = \frac{2x}{\sqrt{\pi}} \sum_{n=0}^{+\infty} (-1)^n \frac{x^{2n}}{(2n+1)\,n!}\,, \tag{1.1}$$

$$\operatorname{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{+\infty} \frac{(2x^2)^n}{1\cdot 3\cdot 5\cdots(2n+1)}\,, \tag{1.2}$$

$$\operatorname{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} \left(1 + \sum_{n=1}^{N-1} (-1)^n \frac{1\cdot 3\cdot 5\cdots(2n-1)}{(2x^2)^n}\right) + \varepsilon_N^{(3)}(x) \tag{1.3}$$

$$\text{où } |\varepsilon_N^{(3)}(x)| \le \frac{e^{-x^2}}{x\sqrt{\pi}} \cdot \frac{1\cdot 3\cdot 5\cdots(2N-1)}{(2x^2)^N}. \tag{1.4}$$

On peut écrire ces formules de façon plus compacte en utilisant le symbole ${}_1F_1$ des fonctions hypergéométriques :

$${}_1F_1(a,\,b;\,x) = \sum_{i=0}^{+\infty} \frac{a\,(a+1)\,\cdots\,(a+n-1)\,x^n}{b\,(b+1)\,\cdots\,(b+n-1)\,(n!)}.$$

On a alors

$$\operatorname{erf}(x) = \frac{2x}{\sqrt{\pi}}\ {}_1F_1\left(\frac{1}{2},\,\frac{3}{2};\,-x^2\right) \tag{1.1}$$

$$\text{et}\ \ \operatorname{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}}\ {}_1F_1\left(1,\,\frac{3}{2};\,x^2\right). \tag{1.2}$$

L'équation (1.1) est intéressante essentiellement pour de petites valeurs de $x$. La série est alternée de sorte que le reste est majoré par le premier terme négligé. Le rapport entre deux termes consécutifs est approximativement $x^2/n$. Ainsi, lorsque $x < 1$, $x^2$ et $n$ contribuent tous les deux à réduire le terme général ; la convergence est donc rapide. Pour de plus grandes valeurs de $x$, la convergence est assez lente car seule la division par $n$ assure la décroissance du terme général. De toutes façons, le principal inconvénient de la série de Taylor pour de grands arguments ne réside pas dans la rapidité de convergence.

Le principal défaut de (1.1) pour de grands arguments vient du fait que la série est alternée : du coup, la somme est mal conditionnée et beaucoup de précision est perdue lors de son évaluation. Ceci est dû à un phénomène communément appelé *cancellation catastrophique* [For70].

**Définition 1.2** (Cancellation)**.** On dit qu'il y a *cancellation* de chiffres lors de la soustraction de deux nombres $a$ et $b$ lorsque les chiffres de poids fort de $a$ et $b$ s'annulent entre eux et que seuls les bits de poids faible contribuent à calculer l'information pertinente sur le résultat.

Si $a$ et $b$ sont connus exactement, le phénomène est bénin. Mais si $a$ et $b$ sont entachés d'une certaine erreur, la cancellation conduit à une perte plus ou moins importante de précision.

Aussi les calculs doivent être effectués en utilisant une précision intermédiaire bien plus grande que la précision cible. Nous quantifierons précisément ce phénomène à la section 1.5.3 (proposition 1.13 et lemme 1.14) : comme nous le verrons, lorsque $x$ augmente, l'équation (1.1) devient inutilisable en pratique.

L'équation (1.2) ne présente pas ce problème de cancellation. Son évaluation ne requiert donc pas beaucoup plus de précision que la précision cible. Cependant, sa convergence est un petit peu moins rapide et il faut en plus évaluer $e^{-x^2}$ (la complexité de ce calcul étant approximativement la même que celle de l'évaluation de la série proprement dite).

L'équation (1.3) donne un moyen très efficace d'évaluer erfc et erf pour de grands arguments. Cependant $\varepsilon_N^{(3)}(x)$ ne peut pas être rendu arbitrairement petit en augmentant $N$ (il y a une valeur de $N$ optimale atteinte en $\lfloor x^2 + 1/2 \rfloor$). Si $\operatorname{erfc}(x)$ doit être évalué avec une précision plus grande, on est obligé d'en revenir à l'équation (1.1) ou (1.2).

### 1.3.1 Schéma d'évaluation

Les trois sommes ont la même structure globale : ce sont des polynômes ou des séries (en la variable $x^2$, $2x^2$ ou $1/(2x^2)$) dont les coefficients sont donnés par une récurrence simple impliquant des multiplications et des divisions par des entiers.

Il faut noter que les entiers en question tiendront en pratique sur des entiers machine de 32 ou 64 bits. Or il est connu [Bre78] que la multiplication (ou la division) d'un nombre flottant de $t$ bits par un entier machine peut être effectuée en temps $\mathcal{O}(t)$. Cette complexité est à comparer avec la complexité d'une multiplication entre deux nombres de $t$ bits qui est $\mathcal{O}(t \log(t) \log(\log(t)))$ asymptotiquement (et seulement quadratique en pratique pour des précisions petites). Les additions et soustractions de deux nombres flottants de $t$ bits s'effectuent aussi en temps $\mathcal{O}(t)$. Le coût total de l'évaluation de la somme est donc globalement dominé par le nombre de multiplications en grande précision (c'est-à-dire entre deux nombres de $t$ bits).

Nous pouvons tirer parti de la structure particulière des sommes et réduire ainsi le nombre de multiplications en grande précision. Cette technique, introduite initialement par Paterson et Stockmeyer [PS73], est présentée par Smith [Smi89] comme le *principe de sommation concurrente d'une série*. Supposons que nous voulions évaluer le polynôme suivant (en la variable $y$) :

$$S(y) = \alpha_0 + \alpha_0\alpha_1 y + \cdots + \alpha_0\alpha_1 \cdots \alpha_{N-1} y^{N-1}. \tag{1.5}$$

Nous supposons que les $\alpha_i$ ($i = 0, \ldots, N-1$) sont des petits entiers ou des inverses d'entier. La multiplication par $\alpha_i$ est donc rapide. On suppose que $L$ est un paramètre entier, et par souci de simplicité on suppose que $N$ est un multiple de $L$. Le cas général se déduit aisément de ce cas particulier. Smith a remarqué que $S(y)$ s'exprime de la façon suivante :

$$\begin{array}{rcl}
S(y) &=& 1 \quad ( \quad \alpha_0 \quad + \quad \alpha_0 \cdots \alpha_L (y^L) \quad + \quad \cdots \quad + \quad \alpha_0 \cdots \alpha_{N-L} (y^L)^{N/L-1} \quad ) \\
&+& y \quad ( \quad \alpha_0\alpha_1 \quad + \quad \alpha_0 \cdots \alpha_{L+1} (y^L) \quad + \quad \cdots \quad + \quad \alpha_0 \cdots \alpha_{N-L+1} (y^L)^{N/L-1} \quad ) \\
&+& \cdots \\
&+& y^{L-1} \quad ( \quad \alpha_0 \cdots \alpha_{L-1} \quad + \quad \alpha_0 \cdots \alpha_{2L-1} (y^L) \quad + \quad \cdots \quad + \quad \alpha_0 \cdots \alpha_{N-1} (y^L)^{N/L-1} \quad ).
\end{array}$$

En notant $S_i$ la somme entre parenthèses sur la $i$-ème ligne de ce tableau, on a donc

$$S(y) = S_0 + S_1 y + \cdots + S_{L-1} y^{L-1}.$$

Les sommes $S_i$ sont évaluées de façon concurrente : une variable acc est utilisée pour stocker les valeurs successives

$$\alpha_0, \quad \alpha_0\alpha_1, \quad \ldots \quad , \quad \alpha_0 \cdots \alpha_{L-1}, \quad \alpha_0 \cdots \alpha_L(y^L), \quad \text{etc.}$$

et les sommes $S_i$ sont calculées au fur et à mesure. La puissance $y^L$ est calculée uniquement une fois, au début, en temps $\mathcal{O}(\log(L))$. Les multiplications (ou divisions) sont toutes des multiplications rapides sauf la multiplication par $y^L$ qui a lieu $N/L - 1$ fois.

Finalement, le polynôme $S(y) = S_0 + S_1 y + \cdots + S_{L-1}y^{L-1}$ est évalué par le schéma de Horner qui nécessite $L - 1$ multiplications en grande précision.

```
1  Algorithme : ConcurrentSeries
   Input : y, L, N, t
   Output : la somme S(y) de l'équation (1.5)
   /* chaque opération est effectuée en précision t                    */
2  z ← power(y, L) ;                    /* obtenu par exponentiation binaire */
3  S ← [0, ..., 0] ;                    /* tableau de L nombres flottants */
4  acc ← 1 ;
5  i ← 0 ;                                          /* indique le S_i courant */
6  for k ← 0 to N − 1 do
7  |  acc ← acc ∗ α_k ;
8  |  S[i] ← S[i] + acc ;
9  |  if i = L − 1 then
10 |  |  i ← 0 ;
11 |  |  acc ← acc ∗ z ;
12 |  else
13 |  |  i ← i + 1 ;
14 |  end
15 end
   /* À présent S(y) est évaluée à partir des S_i par schéma de Horner  */
16 R ← S[L − 1] ;
17 for i ← L − 2 downto 0 do
18 |  R ← S[i] + y ∗ R ;
19 end
20 return R;
```

**Algorithme 1.1 :** Algorithme de sommation concurrente d'une série

L'algorithme complet nécessite $N/L + L - 2 + \mathcal{O}(\log(L))$ multiplications en grande précision (et $\mathcal{O}(N)$ additions et multiplications/divisions par des petits entiers). La valeur optimale de $L$ est $L \simeq \sqrt{N}$. Le coût total est alors approximativement $2\sqrt{N}$ multiplications lentes alors qu'une évaluation directe de toute la somme par schéma de Horner aurait nécessité environ $N$ multiplications lentes. Il faut remarquer toutefois que cette méthode nécessite de l'espace mémoire pour stocker les valeurs intermédiaires $S_i$ jusqu'à l'évaluation finale ($Lt$ bits sont donc nécessaires). L'algorithme complet est résumé en figure « algorithme 1.1 ».

Finissons par une dernière remarque sur cet algorithme. Considérons la valeur de la variable acc juste après que la ligne 7 de l'algorithme a été exécutée. La variable acc est alors une valeur approchée de $\alpha_0 \cdots \alpha_k (y^L)^{\lfloor k/L \rfloor}$. Or, à tout moment, $i = (k \mod L)$ donc $(\mathsf{acc} \cdot y^i)$ est une valeur approchée de $\alpha_0 \cdots \alpha_k y^k$ qui est le coefficient d'ordre $k$ du polynôme $S(y)$. Cela signifie qu'il n'est

pas forcément nécessaire de connaître $N$ en avance : on peut utiliser un test de la forme *sommer les termes jusqu'à en trouver un dont la valeur absolue est inférieure à une certaine borne*. Bien sûr, $(\mathsf{acc} \cdot y^i)$ n'est qu'une valeur approchée du coefficient exact. Si l'on n'est pas prudent, on risque de sous-estimer la valeur exacte et d'arrêter la sommation trop tôt. Il nous faudra donc choisir les modes d'arrondis lors de l'évaluation de la somme de telle sorte que $\mathsf{acc}$ reste toujours une surestimation de la valeur absolue exacte.

## 1.4 Analyse d'erreur

Puisque les opérations de l'algorithme 1.1 sont effectuées en arithmétique flottante, elles sont fondamentalement inexactes et les erreurs d'arrondi doivent donc être prises en compte. Il nous faut choisir avec soin une précision de travail $t$ qui permettra de garder les erreurs d'arrondi suffisamment petites. Des techniques bien connues permettent de borner rigoureusement ces erreurs d'arrondi en fonction de $t$. Dans son livre *Accuracy and Stability of Numerical Algorithms* [Hig02], Higham explique en détail ce que l'honnête homme est supposé savoir en ce domaine. Nous rappelons ici les définitions et propriétés essentielles. Le lecteur intéressé se reportera à [Hig02] pour plus de détails.

**Définition 1.3.** Si $x \in \mathbb{R}$, nous notons $\diamond(x)$ un arrondi fidèle de $x$. Si $t$ est la précision courante, la quantité $u = 2^{1-t}$ est appelée *l'unité d'arrondi*.
Nous utiliserons la notation [2] $z = z' \langle k \rangle$ pour dire que

$$\exists \delta_1, \ldots, \delta_k \in \mathbb{R},\ s_1, \ldots, s_k \in \{-1, 1\},\ \text{ tels que } z = z' \prod_{i=1}^{k} (1+\delta_i)^{s_i} \quad \text{ avec } |\delta_i| \leq u.$$

Remarquons que pour tous $k$ et $k'$ tels que $k' \geq k$, si l'on peut écrire $z' = z \langle k \rangle$, alors on peut aussi écrire $z' = z \langle k' \rangle$.

Cette notation correspond à l'accumulation de $k$ erreurs successives durant les calculs. La proposition suivante en justifie l'usage :

**Proposition 1.4.** *Pour tout $x \in \mathbb{R}$, il existe $\delta \in \mathbb{R}$, $|\delta| \leq u$ tel que $\diamond(x) = x(1+\delta)$. Si $\oplus$ désigne l'addition avec arrondi correct, on a :*

$$\forall (x, y) \in \mathbb{R}^2,\ x \oplus y = \diamond(x+y) = (x+y)(1+\delta) \quad \textit{pour un certain } |\delta| \leq u.$$

*La même chose vaut bien sûr pour les autres opérations correctement arrondies $\ominus$, $\otimes$, $\oslash$, etc.*

À présent, réalisons une analyse d'erreur complète sur un petit exemple afin de comprendre comment on s'y prend. Considérons pour cela six nombres flottants $a$, $b$, $c$, $d$, $e$ et $f$. On veut calculer la somme $S = ab + cde + f$. En pratique on calculera par exemple $\widehat{S} = ((a \otimes b) \oplus ((c \otimes d) \otimes e)) \oplus f$. Les erreurs d'arrondi sont évaluées de la façon suivante :

$$\begin{aligned}
\widehat{S} &= ((a \otimes b) \oplus ((c \otimes d) \otimes e)) \oplus f \\
&= ((ab)\langle 1 \rangle \oplus (cde)\langle 2 \rangle) \oplus f \\
&= ((ab)\langle 2 \rangle \oplus (cde)\langle 2 \rangle) \oplus f \\
&= ((ab)\langle 2 \rangle + (cde)\langle 2 \rangle\langle 1 \rangle) \oplus f \\
&= ((ab)\langle 3 \rangle + (cde)\langle 3 \rangle) \oplus f \\
&= (ab)\langle 4 \rangle + (cde)\langle 4 \rangle + f\langle 4 \rangle .
\end{aligned}$$

Nous avons alors besoin d'une nouvelle proposition.

2. D'après [Hig02], cette notation est introduite par G. W. Stewart sous le nom de *relative error counter*.

**Proposition 1.5.** *Soit $z \in \mathbb{R}$ et $z'$ un nombre flottant. Nous supposons que $k \in \mathbb{N}$ vérifie $ku < 1$ et qu'on peut écrire $z' = z\,\langle k \rangle$. Alors*

$$\exists \theta_k \in \mathbb{R},\ \textit{tel que } z' = z(1+\theta_k) \qquad \textit{avec } |\theta_k| \leq \gamma_k = \frac{ku}{1-ku}.$$

*En particulier, dès que $ku \leq 1/2$, $\quad \gamma_k \leq 2ku$.*

En utilisant cette proposition, nous pouvons revenir à notre exemple et écrire

$$\widehat{S} = (ab)(1+\theta_4) + (cde)(1+\theta'_4) + f(1+\theta''_4).$$

Enfin, on obtient $\left|\widehat{S} - S\right| \leq \gamma_4 \left(|ab| + |cde| + |f|\right)$.

On remarquera l'importance de $\widetilde{S} = |ab| + |cde| + |f|$. Si les termes de la somme sont tous positifs, $S = \widetilde{S}$ et l'erreur relative pour le calcul de la somme est bornée par $\gamma_4$. Si certains termes sont négatifs, l'erreur relative est bornée par $\gamma_4 \widetilde{S}/S$. Le rapport $\widetilde{S}/S$ peut être extrêmement grand : cela quantifie le phénomène de cancellation, lorsque les termes de la somme s'annulent les uns les autres alors que les erreurs s'accumulent.

## 1.5 Implémentation pratique

### 1.5.1 Schéma général

L'implémentation de chacune des trois formules suivra le même schéma général. Les entrées de l'algorithme sont le point $x \in \mathbb{R}$ où erf ou erfc doivent être évalués, ainsi que la précision cible $t'$.

Quelle que soit la formule utilisée, nous devons tronquer la somme à un certain ordre $N-1$. Nous nous trouvons donc face à deux erreurs de nature différente :

- l'erreur d'approximation : elle vient du fait qu'on ignore le reste de la série (l'équation (1.3) n'est pas vraiment une série, mais $\varepsilon_N^{(3)}(x)$ joue le rôle de reste) ;
- les erreurs d'arrondi : elles viennent de l'accumulation d'erreurs durant les calculs en virgule flottante utilisés pour calculer la somme partielle.

L'erreur relative finale doit être plus petite que $2^{-t'}$, autrement dit l'erreur absolue doit être inférieure à $2^{-t'}\operatorname{erf}(x)$ (ou $2^{-t'}\operatorname{erfc}(x)$ pour erfc). Nous coupons cette erreur en deux parties égales et choisissons un rang de troncature $N$ et une précision de travail $t$ de telle sorte que l'erreur d'approximation aussi bien que l'erreur d'arrondi soient inférieures à $2^{-t'-1}\operatorname{erf}(x)$ (ou $2^{-t'-1}\operatorname{erfc}(x)$).

L'erreur d'approximation est contrôlée par le rang de troncature. Supposons que nous connaissons un majorant $\varepsilon_N$ du reste et une minoration positive $f(x)$ de $\operatorname{erf}(x)$ (ou $\operatorname{erfc}(x)$). Il suffit alors de choisir $N$ de telle sorte que $\varepsilon_N \leq 2^{-t'-1} f(x)$.

Dans un premier temps, nous inversons cette formule et obtenons donc une surestimation grossière de $N$. Cette estimation n'a pas besoin d'être très précise : nous l'utiliserons uniquement pour choisir le paramètre $L$ de l'algorithme 1.1 et pour choisir la précision de travail. Lorsque l'algorithme 1.1 est effectivement exécuté, on fait tourner la boucle sur $k$ jusqu'à ce que $\varepsilon_k \leq 2^{-t'-1} f(x)$ ($\varepsilon_k$ sera facilement exprimé en fonction du coefficient d'ordre $k$ de la série). Ainsi, on ne somme finalement qu'un certain nombre $N^\star$ de termes.

Supposons par exemple que $N$ surestime $N^\star$ d'un facteur 4. En choisissant $L \simeq \sqrt{N} = \sqrt{4N^\star}$ on effectuera en fin de compte $N^\star/L + L \simeq 5\sqrt{N^\star}/2$ multiplications lentes, ce qui n'est pas si loin de l'optimal. Comme nous le verrons, seul le logarithme de $N$ importe dans la détermination de la précision de travail $t$. Ainsi, une surestimation d'un facteur 4 conduira à utiliser simplement 2 bits de plus que nécessaire, ce qui est insignifiant.

Dans un second temps, nous effectuons l'analyse d'erreur de l'algorithme (en suivant la même démarche que celle présentée sur l'exemple de la section 1.4). On obtient ainsi une erreur d'arrondi finale, qui sera typiquement de la forme

$$|S - \widehat{S}| \leq \gamma_{aN} \widetilde{S}$$

où $a$ est un entier (dont l'ordre de grandeur sera compris entre 1 et 10 approximativement), $S$ est la somme exacte, $\widehat{S}$ est la somme calculée et $\widetilde{S}$ est la somme des valeurs absolues. Ainsi, il suffit de choisir $t$ de telle sorte que $(2aN)\, 2^{1-t}\, \widetilde{S} \leq 2^{-t'-1} f(x)$ ou de façon équivalente

$$t \geq t' + 3 + \log_2(a) + \log_2(N) + \log_2(\widetilde{S}) - \log_2(f(x)).$$

### 1.5.2 Résultats techniques importants

Avant de donner les détails de notre implémentation des équations (1.1), (1.2) et (1.3), nous avons besoin d'un certain nombre de lemmes techniques qui nous permettront d'effectuer rigoureusement les deux étapes que nous venons d'esquisser.

**Lemme 1.6** (Propagation des erreurs à travers une racine carrée)**.** *Soit $k \in \mathbb{N}^\star$, $z$ et $z'$ deux nombres tels qu'on peut écrire $z' = z\,\langle k \rangle$. Alors, on peut écrire $\sqrt{z'} = \sqrt{z}\,\langle k \rangle$.*

*Démonstration.* Laissée au lecteur. □

**Lemme 1.7** (Propagation des erreurs à travers exp)**.** *Soit $z \in \mathbb{R}$. On note $E$ son exposant : $2^{E-1} \leq z < 2^E$. Soit $t$ une précision. On note $y = z^2$ et on suppose que $\widehat{y} = \diamond(z^2)$, l'opération étant effectuée avec une précision supérieure ou égale à $t + 2E$. Alors*

$$e^{-\widehat{y}} = e^{-z^2}(1+\delta)^s \quad \text{où } |\delta| \leq 2^{1-t} \text{ et } s \in \{-1, 1\}.$$

*En d'autres termes, on peut écrire $e^{-\widehat{y}} = e^{-z^2}\,\langle 1 \rangle$ en précision $t$.*

*Démonstration.* Laissée au lecteur. □

Pour borner le reste de la série, on a besoin de connaître une valeur approchée de $n!$. Nous utilisons les estimations suivantes :

**Lemme 1.8** (Estimation grossière de $n!$)**.** *Les inégalités suivantes sont vraies pour tout $n \geq 1$ :*

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq e\sqrt{n}\left(\frac{n}{e}\right)^n.$$

*Démonstration.* Considérons la suite définie pour $n \geq 1$ par

$$u_n = \frac{\sqrt{2\pi n}(n/e)^n}{n!}.$$

Cette suite est croissante (comme on s'en aperçoit en considérant $u_{n+1}/u_n$). Il est bien connu (formule de Stirling) que $u_n \to 1$ lorsque $n \to +\infty$. Ainsi, pour tout $n \geq 1$, $u_1 \leq u_n \leq 1$. Ceci fournit le résultat annoncé. □

L'estimation grossière de $N$ sera obtenue en inversant une certaine relation. Cette relation implique la fonction $v \mapsto v \log_2(v)$. Nous allons donc avoir besoin d'estimer sa réciproque $\varphi$. La fonction $\varphi$ est apparentée à la fonction $W$ de Lambert (définie comme la réciproque de $x \mapsto x\, e^x$) qui a été bien étudiée [CGH+96, HH08]. Cependant, les bornes connues pour la fonction $W$ de Lambert ne nous permettent pas de majorer et minorer $\varphi$ de façon suffisamment fine sur tout son domaine. Les deux lemmes ci-dessous fournissent des bornes adaptées à notre problème.

**Lemme 1.9** (Inverse de $v\log_2(v)$)**.** *La fonction $v \mapsto v\log_2(v)$ est croissante pour $v \geq 1/e$. On note $\varphi$ son inverse : $\varphi : w \mapsto \varphi(w)$ définie pour $w \geq -\log_2(e)/e$ et telle que pour tout $w$, $\varphi(w)\log_2(\varphi(w)) = w$. La fonction $\varphi$ est croissante.*

*Les inégalités suivantes sont vraies :*

$$\begin{array}{lrcccl} \textit{si } w \in [-\log_2(e)/e,\, 0], & 2^{ew} & \leq & \varphi(w) & \leq & 2^w\ ; \\ \textit{si } w \in [0,\, 2], & 2^{w/2} & \leq & \varphi(w) & \leq & 2^{1/4}\, 2^{w/2}\ ; \\ \textit{si } w \geq 2, & w/\log_2(w) & \leq & \varphi(w) & \leq & 2w/\log_2(w). \end{array}$$

*Démonstration.* Prouver que $v \mapsto v\log_2(v)$ est croissante pour $v \geq 1/e$ est facile et laissé au lecteur. Nous nous contentons de prouver la deuxième inégalité ; les autres s'obtiennent en utilisant la même technique.

On désigne $\varphi(w)$ par $v$ : si $w \in [0, 2]$, il est aisé de vérifier que $v \in [1, 2]$. À présent, puisque $\log_2(v) = w/v$, on obtient $w/2 \leq \log_2(v) \leq w$. Et donc

$$2^{w/2} \leq v \leq 2^w, \tag{1.6}$$

ce qui fournit la minoration.

On peut raffiner cette identité : en utilisant encore le fait que $\log_2(v) = w/v$, on obtient

$$\frac{w}{2^w} \leq \log_2(v) \leq \frac{w}{2^{w/2}} \quad \text{et donc} \quad v \leq 2^{(w\,2^{-w/2})}.$$

Pour terminer la preuve, nous devons simplement montrer que, pour tout $w \in [0, 2]$,

$$w\,2^{-w/2} \leq 1/4 + w/2.$$

Le développement de Taylor de $2^{-w/2}$ est

$$2^{-w/2} = \sum_{n=0}^{+\infty} (-1)^n \frac{(w\ln(2))^n}{2^n\, n!}.$$

La série est alternée et de terme général décroissant quand $w \in [0, 2]$. De là,

$$2^{-w/2} \leq 1 - \frac{w\ln(2)}{2} + \frac{w^2\ln(2)^2}{8}.$$

À partir de cette inéquation, on a

$$w\,2^{-w/2} - \frac{w}{2} \leq w\left(\frac{1}{2} - \frac{w\ln(2)}{2} + \frac{w^2\ln(2)^2}{8}\right).$$

En étudiant les variations du membre droit de cette inéquation on arrive finalement à

$$w\,2^{-w/2} - \frac{w}{2} \leq \frac{4}{27\ln(2)} \simeq 0.2137\cdots \leq 1/4$$

ce qui conclut la preuve.

**Remarque :** la dernière inégalité se montre en utilisant le fait que $v \leq v\log_2(v) \leq v^2$ pour $v \geq 2$. Cette inégalité se récrit $\sqrt{w} \leq v \leq w$ à partir de quoi on utilise la même technique. □

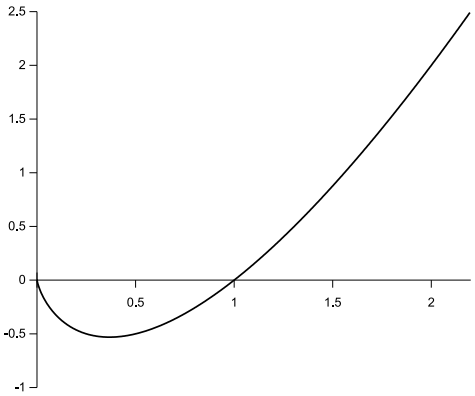



**Figure 1.5 :** Graphe de la fonction $v \mapsto v \log_2(v)$

**Lemme 1.10** (Inverse de $v \log_2(v)$, l'autre branche)**.** *La fonction $v \mapsto v \log_2(v)$ est décroissante pour $0 \leq v \leq 1/e$. On note $\varphi_2$ son inverse : $\varphi_2 : w \mapsto \varphi_2(w)$ telle que $\varphi_2(w) \log_2(\varphi_2(w)) = w$. La valeur $\varphi_2(w)$ est définie pour $-\log_2(e)/e \leq w \leq 0$ et elle est décroissante.*

*Les inégalités suivantes donnent une estimation de $\varphi_2(w)$ :*

$$\forall w \in \left[-\frac{\log_2(e)}{e}, 0\right[, \qquad \frac{1}{3} \cdot \frac{w}{\log_2(-w)} \;\leq\; \varphi_2(w) \;\leq\; \frac{w}{\log_2(-w)}.$$

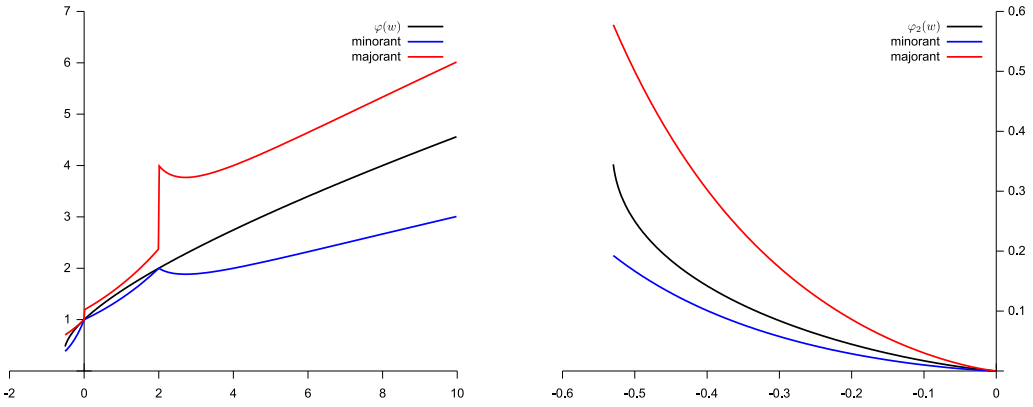
*Démonstration.* Prouver que $v \mapsto v \log_2(v)$ est décroissante pour $v \in [0, 1/e]$ est aisé et laissé au lecteur. Nous utilisons les notations suivantes qui sont plus pratiques :

$$\begin{aligned} \omega &= 1/(-w), \\ \nu &= 1/\varphi_2(w). \end{aligned}$$

Ainsi $\omega \geq e/\log_2(e)$ et $\nu \geq e$. De plus, par hypothèse, $\nu/\log_2(\nu) = \omega$. Il est aisé de voir que pour tout $\nu \geq e$,

$$\nu^{1/3} \leq \frac{\nu}{\log_2(\nu)} \leq \nu.$$

De là, $\omega \leq \nu \leq \omega^3$ et donc $\log_2(\omega) \leq \log_2(\nu) \leq 3 \log_2(\omega)$. On conclut en utilisant $\nu = \omega \log_2(\nu)$. □




(a) Graphe de $\varphi$ et ses bornes (b) Graphe de $\varphi_2$ et ses bornes

**Figure 1.6 :** Illustration des lemmes 1.9 et 1.10

Pour borner les erreurs relatives, il nous faut des estimations des valeurs de $\operatorname{erf}(x)$ et $\operatorname{erfc}(x)$. Le lemme suivant donne de telles estimations :

**Lemme 1.11.** *Les inégalités suivantes sont vraies :*

$$\begin{array}{lrcccl} \textit{si } 0 < x < 1, & x/2 & \leq & \operatorname{erf}(x) & \leq & 2x, \\ & 1/8 & \leq & \operatorname{erfc}(x) & \leq & 1\,; \\ \\ \textit{si } x \geq 1, & 1/2 & \leq & \operatorname{erf}(x) & \leq & 1, \\ & e^{-x^2}/(4x) & \leq & \operatorname{erfc}(x) & \leq & e^{-x^2}/(x\sqrt{\pi}). \end{array}$$

*Démonstration.* Lorsque $0 < x < 1$, la série donnée par l'équation (1.1) est alternée et de terme général décroissant. Donc

$$\frac{2x}{\sqrt{\pi}}\left(1 - \frac{x^2}{3}\right) \leq \operatorname{erf}(x) \leq \frac{2x}{\sqrt{\pi}}.$$

Les inégalités pour $\operatorname{erf}(x)$ s'en déduisent facilement.

Puisque erfc est décroissante, $\operatorname{erfc}(1) \leq \operatorname{erfc}(x) \leq \operatorname{erfc}(0) = 1$. En prenant un terme de plus dans la série de $\operatorname{erf}(x)$, on obtient

$$\operatorname{erf}(x) \leq \frac{2x}{\sqrt{\pi}}\left(1 - \frac{x^2}{3} + \frac{x^4}{10}\right).$$

En l'appliquant en $x = 1$, on obtient $\operatorname{erfc}(1) = 1 - \operatorname{erf}(1) \geq 1/8$.

Lorsque $x > 1$, et puisque erf est croissante et tend vers 1 en $+\infty$, on a $\operatorname{erf}(1) \leq \operatorname{erf}(x) \leq 1$. Cela fournit les inégalités pour $\operatorname{erf}(x)$.

Nous prouvons à présent les deux dernières inégalités du lemme. Par définition,

$$\begin{aligned} \operatorname{erfc}(x) &= \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-v^2}\,\mathrm{d}v \\ &= \frac{2}{\sqrt{\pi}} \int_x^{+\infty} \frac{-2v\,e^{-v^2}}{-2v}\,\mathrm{d}v \\ &= \frac{2}{\sqrt{\pi}} \left(\frac{e^{-x^2}}{2x} - \int_x^{+\infty} \frac{e^{-v^2}}{2v^2}\,\mathrm{d}v\right). \end{aligned}$$

Cela fournit la majoration. Pour la minoration, on utilise le changement de variable $v \leftarrow v + x$ :

$$\int_x^{+\infty} \frac{e^{-v^2}}{2v^2}\,\mathrm{d}v = \int_0^{+\infty} \frac{e^{-x^2-2vx-v^2}}{2(v+x)^2}\,\mathrm{d}v.$$

Puisque $(v+x)^2 \geq x^2$ et $-v^2 \leq 0$,

$$\int_0^{+\infty} \frac{e^{-x^2-2vx-v^2}}{2(v+x)^2}\,\mathrm{d}v \leq \frac{e^{-x^2}}{2x^2} \int_0^{+\infty} e^{-2vx}\,\mathrm{d}v = \frac{e^{-x^2}}{4x^3}.$$

De là,

$$\begin{aligned} \operatorname{erfc}(x) &\geq \frac{e^{-x^2}}{x\sqrt{\pi}}\left(1 - \frac{1}{2x^2}\right) \\ &\geq \frac{e^{-x^2}}{2x\sqrt{\pi}} \quad \text{puisque } x \geq 1. \end{aligned}$$

Nous concluons en remarquant que $2\sqrt{\pi} \simeq 3.544 \leq 4$. □

### 1.5.3 Implémentation de l'équation (1.1) en pratique

Dans cette section, nous étudions comment utiliser l'équation (1.1) pour obtenir une valeur approchée de $\operatorname{erf}(x)$ (rappelons qu'on suppose $x > 0$) :

$$\operatorname{erf}(x) = \frac{2x}{\sqrt{\pi}} \left( \sum_{n=0}^{N-1} (-1)^n \frac{(x^2)^n}{(2n+1)n!} \right) + \varepsilon_N^{(1)}(x)$$

où $\varepsilon_N^{(1)}(x)$ est le reste.

Nous commençons par trouver une relation qui assure que $\varepsilon_N^{(1)}(x)$ est inférieur à $2^{-t'-1} \operatorname{erf}(x)$ (pour rappel, $t'$ est la précision cible, qui est un paramètre de l'algorithme).

**Proposition 1.12.** *Soit $E$ l'exposant de $x$. Si $N$ vérifie*

$$\frac{N}{ex^2} \log_2 \left( \frac{N}{ex^2} \right) \geq \frac{t' + \max(0,\, E)}{ex^2},$$

*le reste est borné par $2^{-t'-1} \operatorname{erf}(x)$.*

*Démonstration.* Déjà, remarquons que pour $N \geq 1$,

$$\frac{2}{\sqrt{\pi}} \cdot \frac{1}{(2N+1)\sqrt{2\pi N}} \leq \frac{1}{4}. \tag{1.7}$$

Nous distinguons deux cas, suivant que $x < 1$ ou $x \geq 1$ :

— si $x < 1$, $E \leq 0$ et $\operatorname{erf}(x)$ est supérieur à $x/2$. L'hypothèse devient donc $(ex^2/N)^N \leq 2^{-t'}$ d'où l'on déduit

$$\frac{x^{2N+1}}{2} \left( \frac{e}{N} \right)^N \leq 2^{-t'} \operatorname{erf}(x); \tag{1.8}$$

— si $x \geq 1$, $E > 0$ et $\operatorname{erf}(x)$ est supérieur à $1/2$. L'hypothèse devient alors $(ex^2/N)^N \leq 2^{-t'-E}$ d'où

$$\frac{x^{2N}}{2} \left( \frac{e}{N} \right)^N \leq 2^{-t'-E} \operatorname{erf}(x).$$

Puisque $x < 2^E$, on obtient bien l'inégalité (1.8) à nouveau.

Des inéquations (1.7) et (1.8) on déduit que

$$\frac{2}{\sqrt{\pi}} \cdot \frac{x^{2N+1}}{2N+1} \left( \frac{e}{N} \right)^N \frac{1}{\sqrt{2\pi N}} \leq 2^{-t'-1} \operatorname{erf}(x).$$

En utilisant le lemme 1.8, on arrive à

$$\frac{2}{\sqrt{\pi}} \cdot \frac{x^{2N+1}}{(2N+1)N!} \leq 2^{-t'-1} \operatorname{erf}(x).$$

À présent, remarquons que le terme de gauche de l'inéquation est la valeur absolue du terme général de la série. Puisque ce terme est inférieur à $2^{-t'-1} \operatorname{erf}(x)$, il est en particulier inférieur à $1/2$. Comme la série est alternée, on peut borner le reste par la valeur absolue du premier terme négligé dès que le terme général décroît en valeur absolue.

Dans notre cas, la valeur absolue du terme général peut commencer par croître avant de décroître. Cependant, lorsqu'elle est croissante, elle est toujours plus grande que 1. Par suite, dans la mesure où ici elle est inférieure à $1/2$, on est dans la phase décroissante et on peut donc affirmer $\varepsilon_N^{(1)}(x) \leq 2^{-t'-1} \operatorname{erf}(x)$. □

Cette proposition et le lemme 1.9 nous permettent d'obtenir une majoration de $N$. Pour cela, on évalue $a = (t' + \max(0,\, E))/(ex^2)$. En réalité, on effectue le calcul en choisissant soigneusement les modes d'arrondi de façon à obtenir une valeur approchée par excès $a_u \geq a$. Ensuite, on raisonne par condition suffisante :

— d'après la proposition 1.12, il suffit que

$$\frac{N}{ex^2}\,\log_2\left(\frac{N}{ex^2}\right) \geq a\ ;$$

— pour cela, il suffit que

$$\frac{N}{ex^2}\,\log_2\left(\frac{N}{ex^2}\right) \geq a_u\ ;$$

— ceci est équivalent à dire que $N/(ex^2) \geq \varphi(a_u)$ ;

— pour cela, il suffit que $N/(ex^2)$ soit supérieur à un majorant de $\varphi(a_u)$. De tels majorants sont fournis par le lemme 1.9.

On peut donc choisir $N$ en utilisant la règle suivante :

| | | | |
|---|---|---|---|
| – si $a_u \geq 2$, | $N$ | $\geq$ | $(ex^2)\cdot 2a_u/\log_2(a_u)$ ; |
| – si $a_u \in [0,\, 2]$, | $N$ | $\geq$ | $(ex^2)\cdot 2^{1/4}\,2^{a_u/2}$. |

Lorsque $N$ est calculé avec ces formules, les modes d'arrondi sont choisis de telle sorte que la valeur calculée soit une surestimation de la valeur exacte de la formule.

Une fois $N$ déterminé, il nous faut choisir une précision de travail $t$. Cette précision dépend des erreurs qui vont se propager durant l'évaluation. Il nous faut donc, dans un premier temps, esquisser les détails de l'évaluation.

La somme

$$S(x) = \sum_{n=0}^{N-1} (-1)^n \frac{2}{\sqrt{\pi}} \cdot \frac{x^{2n+1}}{(2n+1)\,n!}$$

est évaluée en utilisant l'algorithme 1.1 avec les paramètres $y = x^2$, $\alpha_0 = 2x/\sqrt{\pi}$ et, pour $k \geq 1$, $\alpha_k = \dfrac{-1}{k} \cdot \dfrac{2k-1}{2k+1}$. Dans les paragraphes suivants, les variables acc, tmp, $i$, $k$, etc. sont celles introduites dans l'algorithme 1.1 en page 42.

En pratique, on n'effectue pas la division $(2k-1)/(2k+1)$. On utilise la variable acc pour calculer $(y^L)^{\lfloor k/L \rfloor}/k!$ et une variable temporaire tmp pour effectuer la division par $2k+1$. La variable $S_i$ est mise à jour en lui ajoutant et soustrayant successivement la valeur tmp (au lieu de acc).

Au début de l'algorithme, $y = x^2$ et $z = y^L$ sont calculés en arrondi par excès. Le calcul de $\alpha_0$ est effectué avec les arrondis choisis de telle sorte que la valeur calculée soit une surestimation de la valeur exacte $2x/\sqrt{\pi}$. Les variables acc et tmp sont mises à jour en arrondi par excès. Ainsi l'inégalité suivante reste toujours vraie :

$$\mathsf{tmp} \cdot y^i \geq \frac{2}{\sqrt{\pi}} \cdot \frac{x^{2k+1}}{(2k+1)k!}.$$

Soit $F$ l'exposant de $y$ : $y < 2^F$. En utilisant le fait que $\operatorname{erf}(x) \geq x/2$ lorsque $x < 1$ et $\operatorname{erf}(x) \geq 1/2$ lorsque $x \geq 1$, on voit facilement que la boucle peut être arrêtée dès que

$$k \geq N \quad \text{ou} \quad \mathsf{tmp} \cdot 2^{Fi} < 2^{-t'+\min(E-1,\,0)-2}.$$

L'algorithme complet est résumé en figure « algorithme 1.2 ».

Les erreurs d'arrondi sont bornées à l'aide de la proposition suivante :

```
1  Algorithme : erfByEquation1
   Input : un nombre flottant x,
           la précision courante t,
           la précision cible t',
           L ∈ ℕ*, N ∈ ℕ*.
   Output : une valeur approchée de erf(x) dont l'erreur relative est inférieure à 2^{-t'-1},
            obtenue à partir de l'équation (1.1)
   /* toutes les opérations sont effectuées en précision t                    */
2  y ← x * x ; // arrondi par excès
3  F ← exponent(y) ;
4  if x < 1 then G ← exponent(x) − 1 else G ← 0 ;
5  z ← power(y, L) ; // arrondi par excès
6  S ← [0, ..., 0] ;
7  acc ← √π ; // arrondi par défaut
8  acc ← 2 * x/acc ; // arrondi par excès
9  i ← 0 ;
10 k ← 0 ;
11 tmp ← acc ;
12 repeat
13 |  if (k mod 2) = 0 then S[i] ← S[i] + tmp else S[i] ← S[i] − tmp ;
14 |  k ← k + 1;
15 |  if i = L − 1 then
16 |  |  i ← 0 ;
17 |  |  acc ← acc * z ; // arrondi par excès
18 |  else
19 |  |  i ← i + 1 ;
20 |  end
21 |  acc ← acc/k ; // arrondi par excès
22 |  tmp ← acc/(2 * k + 1) ; // arrondi par excès
23 until k = N or exponent(tmp) < G − t' − 2 − F * i ;
   /* À présent S(y) est calculé à partir des S_i par schéma de Horner       */
24 R ← S[L − 1] ;
25 for i ← L − 2 downto 0 do
26 |  R ← S[i] + y * R ;
27 end
28 return R;
```

**Algorithme 1.2 :** Évaluation de la fonction erf par l'équation (1.1)

**Proposition 1.13.** *Si l'algorithme 1.2 est utilisé pour calculer une valeur approchée $\widehat{S(x)}$ de la somme $S(x)$, alors on peut écrire*

$$\widehat{S(x)} = \sum_{n=0}^{N-1} (-1)^n \frac{2x}{\sqrt{\pi}} \cdot \frac{x^{2n}}{(2n+1)\,n!} \langle 8N \rangle .$$

*Par suite,*

$$\left| \widehat{S(x)} - S(x) \right| \leq \gamma_{8N} \left( \frac{2x}{\sqrt{\pi}} \sum_{n=0}^{N-1} \frac{x^{2n}}{(2n+1)\,n!} \right) \leq \gamma_{8N} \frac{2}{\sqrt{\pi}} \int_0^x e^{v^2} \,\mathrm{d}v.$$

*La borne $\gamma_{8N}$ pourrait être affinée. Cependant, on ne peut espérer obtenir une meilleure borne que $\gamma_N$ puisque l'algorithme implique $\mathcal{O}(N)$ opérations. Comme nous le verrons, seul le logarithme de cette valeur est utile pour choisir la précision de travail $t$. Une analyse plus fine permettrait donc, au mieux, de remplacer* $\log(8N)$ *par* $\log(N)$ *et l'intérêt pratique d'un tel effort serait à peu près nul.*

*Démonstration.* Pour prouver ce résultat, nous utilisons la même technique que celle présentée sur un exemple en section 1.4 page 43. Les arguments majeurs sont les suivants :

- la ligne 2 de l'algorithme conduit à une erreur ;
- la ligne 5 est obtenue par exponentiation binaire : on voit facilement en décomptant les multiplications que $\widehat{z} = z\,\langle 2L-1 \rangle$ ;
- la ligne 7 implique une approximation de $\pi$ ($\widehat{\pi} = \pi\,\langle 1 \rangle$) et une racine carrée. Le lemme 1.6 permet d'obtenir $\mathsf{acc} = \diamond(\sqrt{\widehat{\pi}}) = \diamond(\sqrt{\pi}\,\langle 1 \rangle) = \sqrt{\pi}\,\langle 2 \rangle$ ;
- la ligne 8 nécessite une division (la multiplication par 2 est exacte) ;
- la variable $\mathsf{acc}$ est mise à jour aux lignes 17 et 21 de l'algorithme. En ligne 17, $2L$ erreurs sont accumulées ($2L-1$ qui proviennent de $z$, et 1 provenant de la multiplication elle-même). La ligne 21 accumule une erreur. On voit donc facilement que

$$\widehat{\mathsf{acc}} = \mathsf{acc}\,\langle \underbrace{3}_{\text{initialisation}} + \underbrace{N}_{\text{la ligne 21 arrive au plus } N \text{ fois}} + \underbrace{2L\lfloor N/L \rfloor}_{\text{la ligne 17 arrive au plus } \lfloor N/L \rfloor \text{ fois}} \rangle.$$

  On simplifie cette équation en $\widehat{\mathsf{acc}} = \mathsf{acc}\,\langle 3 + 3N \rangle$ ;
- on peut toujours écrire $\widehat{\mathsf{tmp}} = \mathsf{tmp}\,\langle 4 + 3N \rangle$ puisque $\mathsf{tmp}$ est calculé à partir de $\mathsf{acc}$ par une division ($2k+1$ est calculé exactement en arithmétique entière) ;
- finalement, $S[i]$ est obtenu par moins de $N$ additions impliquant la variable $\mathsf{tmp}$ et on peut donc écrire

$$S_i = \mathsf{tmp}_i\,\langle 4N+4 \rangle + \mathsf{tmp}_{i+L}\,\langle 4N+4 \rangle + \cdots + \mathsf{tmp}_{i+N-L}\,\langle 4N+4 \rangle$$

  où $\mathsf{tmp}_k$ représente la valeur de la variable $\mathsf{tmp}$ à l'étape $k$ de l'algorithme ;
- l'évaluation par schéma de Horner en ligne 26 accumule 3 erreurs supplémentaires à chaque étape (l'une vient du fait que $y = x^2\,\langle 1 \rangle$, une autre provient de la multiplication et la dernière provient de l'addition). La boucle complète accumule donc $3(L-1)$ erreurs par terme de la somme. Nous le bornons par $3N-3$ et obtenons donc finalement le résultat avec $\langle 7N+1 \rangle$. Nous bornons cette valeur par $\langle 8N \rangle$.

□

Il ne nous reste plus qu'à voir comment choisir la précision de travail $t$ et nous aurons complètement décrit l'implémentation. La précision de travail est estimée à l'aide du lemme suivant :

**Lemme 1.14.** *On a les inégalités suivantes :*

$$\begin{array}{lccccc} \textit{si } 0 < x < 1, & x & \leq & \displaystyle\int_0^x e^{v^2}\,\mathrm{d}v & \leq & 2x\,; \\ \textit{si } x \geq 1, & \dfrac{1}{e^2}\cdot\dfrac{e^{x^2}}{x} & \leq & \displaystyle\int_0^x e^{v^2}\,\mathrm{d}v & \leq & \dfrac{e^{x^2}}{x}. \end{array}$$

*Démonstration.* Dans la première inégalité, la minoration est obtenue en remplaçant $e^{v^2}$ par 1 dans l'intégrale. La majoration est obtenue en remplaçant $e^{v^2}$ par $e^{0.55^2}$ sur $[0,\, 0.55]$ et par $e$ sur $[0.55,\, 1]$.

Dans la seconde inégalité, la majoration est obtenue en remplaçant $e^{v^2}$ par $e^{vx}$ dans l'intégrale. La minoration est obtenue en considérant l'intégrale restreinte au sous-intervalle $[x - 1/x,\, x]$. □

Finalement, nous pouvons énoncer la règle qui permet de choisir la précision de travail $t$ (on rappelle que $E$ est l'exposant de $x$) :

| | |
|---|---|
| – lorsque $x < 1$, | $t \geq t' + 9 + \lceil \log_2 N \rceil$ ; |
| – lorsque $x \geq 1$, | $t \geq t' + 9 + \lceil \log_2 N \rceil - E + x^2 \log_2(e)$. |

*Démonstration.* Nous montrons le résultat pour la première inéquation. La seconde s'obtient de la même façon. Supposons que $0 < x < 1$ et $t \geq t' + 9 + \lceil \log_2 N \rceil$. Alors

$$2^{-t+8}\, N \leq 2^{-t'-1}.$$

Puisque $\gamma_{8N} \leq 16Nu$ (proposition 1.5), on a

$$2\gamma_{8N}\,(2x) \leq 2^{-t'-1}\,\frac{x}{2}.$$

On conclut en utilisant le fait que $(x/2) \leq \mathrm{erf}(x)$ (lemme 1.11), $(2/\sqrt{\pi}) \leq 2$ et $\displaystyle\int_0^x e^{v^2}\,\mathrm{d}v \leq 2x$ (lemme 1.14). □

En pratique $\log_2(e)$ est remplacé par une valeur précalculée qui l'approche par excès. Le facteur $x^2 \log_2(e)$ qui apparaît lorsque $x \geq 1$ traduit les effets de cancellation qui font que la série est mal conditionnée pour les grandes valeurs de $x$.

### 1.5.4 Implémentation de l'équation (1.2) en pratique

Dans cette section, nous étudions comment utiliser l'équation (1.2) pour obtenir une valeur approchée de $\mathrm{erf}(x)$ (on suppose toujours que $x > 0$) :

$$\mathrm{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \left( \sum_{n=0}^{N-1} \frac{(2x^2)^n}{1 \cdot 3 \cdot 5 \cdots (2n+1)} \right) + \varepsilon_N^{(2)}(x)$$

où $\varepsilon_N^{(2)}(x)$ est le reste de la série.

Nous suivons la même démarche que pour l'équation (1.1). La première chose à faire est donc de borner le reste. Le lemme suivant répond à ce problème :

**Lemme 1.15.** *Si $N \geq 2x^2$, on a l'inéquation suivante :*

$$\varepsilon_N^{(2)}(x) \leq 2\,\frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^n}{1 \cdot 3 \cdots (2N+1)}.$$

*Démonstration.* Par définition,

$$\begin{aligned}\varepsilon_N^{(2)}(x) &= \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=N}^{+\infty} \frac{(2x^2)^n}{1\cdot 3\cdots(2n+1)} \\ &= \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^N}{1\cdot 3\cdots(2N+1)} \left(1 + \frac{2x^2}{2N+3} + \frac{(2x^2)^2}{(2N+3)(2N+5)} + \cdots\right).\end{aligned}$$

Nous le bornons par la série géométrique de raison $(2x^2)/(2N+3)$ et de premier terme égal à 1 :

$$\varepsilon_N^{(2)}(x) \leq \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^N}{1\cdot 3\cdots(2N+1)} \left(1 + \frac{2x^2}{2N+3} + \frac{(2x^2)^2}{(2N+3)(2N+3)} + \cdots\right).$$

Puisque $N \geq 2x^2$, on a $(2x^2)/(2N+3) \leq 1/2$, et la somme de la série est bornée par 2. □

La relation entre $N$ et $t'$ est donnée par la proposition suivante :

**Proposition 1.16.** *Soit $E$ l'exposant de $x$. Si $N$ vérifie $N \geq 2x^2$ et*

$$\frac{N}{ex^2} \log_2\left(\frac{N}{ex^2}\right) \geq \frac{t' + 3 + \max(0,\, E) - x^2 \log_2(e)}{ex^2},$$

*le reste est borné par* $\operatorname{erf}(x)\, 2^{-t'-1}$.

*Démonstration.* Nous utilisons le même genre d'argument que pour la proposition 1.12. Le produit $1\cdot 3\cdots(2N+1)$ est égal à $(2N+1)!/(2^N\, N!)$. On peut donc écrire

$$\left|\varepsilon_N^{(2)}(x)\right| \leq 4\,\frac{xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^N\, N!\, 2^N}{(2N+1)!} \leq 4\,\frac{xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^N\, N!\, 2^N}{(2N)!}.$$

On conclut en utilisant les lemmes 1.8 et 1.11. □

On commence donc par évaluer

$$a = \frac{t' + 3 + \max(0, E) - x^2 \log_2(e)}{ex^2}.$$

De la même façon que dans la section précédente, on effectue le calcul en choisissant soigneusement à chaque fois les modes d'arrondi de façon à obtenir une surestimation $a_u$ de la valeur exacte $a$. Nous en déduisons les formules pour calculer une surestimation de $N$ :

| | | | |
|---|---|---|---|
| – si $a_u \geq 2$, | $N$ | $\geq$ | $(ex^2)\cdot 2a_u/\log_2(a_u)$ ; |
| – si $a_u \in [0,\, 2]$, | $N$ | $\geq$ | $(ex^2)\cdot 2^{1/4}\, 2^{a_u/2}$ ; |
| – sinon, | $N$ | $\geq$ | $(ex^2)\cdot 2^{a_u}$ ; |
| – dans ce dernier cas, si $N < 2x^2$ prendre $N = \lceil 2x^2 \rceil$. | | | |

Seul le cas $a_u < 0$ peut mener à une valeur de $N$ inférieure à $2x^2$. Si c'est le cas, on prend $N = \lceil 2x^2 \rceil$ de façon à garantir l'hypothèse.

La somme

$$S(x) = \sum_{n=0}^{N-1} \frac{2x\, e^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^n}{1\cdot 3\cdots(2n+1)}$$

est évaluée par l'algorithme 1.1 avec les paramètres $y = 2x^2$, $\alpha_0 = 2xe^{-x^2}/\sqrt{\pi}$ et, pour $k \geq 1$, $\alpha_k = 1/(2k+1)$. De même que dans l'implémentation de l'équation (1.1), les arrondis sont faits par excès et un test permet d'arrêter la boucle principale dès que possible. En l'occurrence, le critère devient

$$k \geq N \quad \text{ou} \quad \left(k \geq 2x^2 \ \text{ et } \ \mathsf{acc} \cdot 2^{Fi} < 2^{-t'+\min(E-1,\,0)-3}\right).$$

L'algorithme complet est résumé en figure « algorithme 1.3 ».

Les erreurs d'arrondi sont bornées en utilisant la proposition suivante :

**Proposition 1.17.** *Si l'algorithme 1.3 est utilisé pour calculer une valeur approchée $\widehat{S(x)}$ de la somme $S(x)$, on a :*

$$\widehat{S(x)} = \sum_{n=0}^{N-1} \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^n}{1 \cdot 3 \cdots (2n+1)} \langle 16N \rangle .$$

*Alors*

$$\left|\widehat{S(x)} - S(x)\right| \leq \gamma_{16N}\, S(x) \leq \gamma_{16N}\ \mathrm{erf}(x).$$

*Démonstration.* En fait, en utilisant les mêmes arguments que pour la preuve de la proposition 1.13, on montre que

$$\widehat{S(x)} = \sum_{n=0}^{N-1} \frac{2xe^{-x^2}}{\sqrt{\pi}} \cdot \frac{(2x^2)^n}{1 \cdot 3 \cdots (2n+1)} \langle 7N+3 \rangle .$$

Nous le bornons par $\langle 16N \rangle$ parce qu'il est plus aisé de manipuler des puissances de 2 (car nous prendrons le logarithme binaire de cette valeur).

Il faut noter qu'à la ligne 10 de l'algorithme, $x^2$ est calculé en précision $t + \max(2E, 0)$. Le lemme 1.7 permet alors d'écrire $\widehat{\alpha_0} = \alpha_0 \langle 6 \rangle$. □

Enfin la précision de travail $t$ est donnée par la règle suivante :

$$\boxed{t \geq t' + 7 + \lceil \log_2 N \rceil .}$$

### 1.5.5 Implémentation de erfc avec l'équation (1.1) ou (1.2)

Dans cette section, nous ne supposons plus que $x > 0$.

Puisque $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$, nous pouvons utiliser les algorithmes précédents pour évaluer erfc. Cependant, nous devons prendre garde à deux choses :

— d'abord, la valeur approchée de $\mathrm{erf}(x)$ doit être calculée avec une erreur relative $2^{-s}$ bien choisie. Puisque $\mathrm{erf}(x)$ et $\mathrm{erfc}(x)$ n'ont pas le même ordre de grandeur, $2^{-s}$ n'a aucune raison d'être la même chose que l'erreur relative cible $2^{-t'}$ avec laquelle on désire connaître $\mathrm{erfc}(x)$ ;

— ensuite, et contrairement à erf, erfc n'est pas impaire (ni paire). En particulier, $\mathrm{erfc}(-x)$ et $\mathrm{erfc}(x)$ n'ont pas le même ordre de grandeur et les deux cas doivent donc être traités séparément lors de l'estimation de l'erreur relative.

Pour calculer $\mathrm{erfc}(x)$, on procède en deux étapes : dans un premier temps, on calcule une valeur approchée $R$ de $\mathrm{erf}(x)$ avec une erreur relative majorée par une certaine borne $2^{-s}$ (ce calcul est effectué en utilisant l'un des deux algorithmes présentés aux sections précédentes). Dans un second temps, on calcule $1 \ominus R$ en précision $t' + 3$.

**Lemme 1.18.** *Si $s$ est choisi conformément à la règle suivante, on a $|R - \mathrm{erf}(x)| \leq 2^{-t'-1}\ \mathrm{erfc}(x)$ :*

| | |
|---|---|
| – *lorsque* $x \leq -1$ | $s \geq t' + 1$ ; |
| – *lorsque* $-1 < x < 0$ | $s \geq t' + 2 + E$ ; |
| – *lorsque* $0 \leq x < 1$ | $s \geq t' + 5 + E$ ; |
| – *lorsque* $x \geq 1$ | $s \geq t' + 3 + E + x^2 \log_2(e)$. |

```
1  Algorithme : erfByEquation2
   Input : un nombre flottant x,
           la précision de travail t,
           la précision cible t',
           L ∈ ℕ*, N ∈ ℕ*.
   Output : une valeur approchée de erf(x) dont l'erreur relative est inférieure à 2^(-t'-1)
            obtenue en utilisant l'équation (1.2)
   /* toutes les opérations sont effectuées en précision t                    */
2  y ← 2 * x * x ; // arrondi par excès
3  E ← exponent(x) ;
4  F ← exponent(y) ;
5  if x < 1 then G ← E − 1 else G ← 0 ;
6  z ← power(y, L) ; // arrondi par excès
7  S ← [0, ..., 0] ;
8  acc ← √π ; // arrondi par défaut
9  acc ← 2 * x/acc ; // arrondi par excès
10 tmp ← x * x ; // effectué en précision t + max(2 E, 0), arrondi par défaut
11 tmp ← exp(−tmp) ; // arrondi par excès
12 acc ← acc * tmp ; // arrondi par excès
13 i ← 0 ;
14 k ← 0 ;
15 repeat
16 |  S[i] ← S[i] + acc ;
17 |  k ← k + 1;
18 |  if i = L − 1 then
19 |  |  i ← 0 ;
20 |  |  acc ← acc * z ; // arrondi par excès
21 |  else
22 |  |  i ← i + 1 ;
23 |  end
24 |  acc ← acc/(2 * k + 1) ; // arrondi par excès
25 until k = N or ((k ≥ y) and (exponent(acc) < G − t' − 3 − F * i)) ;
   /* À présent, on évalue S(y) à partir des S_i par schéma de Horner        */
26 R ← S[L − 1] ;
27 for i ← L − 2 downto 0 do
28 |  R ← S[i] + y * R ;
29 end
30 return R;
```

**Algorithme 1.3 :** Évaluation de la fonction erf par l'équation (1.2)

*Démonstration.*
**Premier cas.**
On suppose $x \leq -1$ et $s \geq t' + 1$. À partir des hypothèses, on a

$$|R - \operatorname{erf}(x)| \;\leq\; 2^{-s}\,|\operatorname{erf}(x)| \;\leq\; 2^{-t'-1}\,|\operatorname{erf}(x)|.$$

Puisque $x < 0$, $\operatorname{erfc}(x) \geq 1$. De plus, $|\operatorname{erf}(x)| \leq 1$. Cela fournit le résultat.

**Deuxième cas.**
On suppose $-1 < x < 0$ et $s \geq t' + 2 + E$. À partir des hypothèses, on a

$$|R - \operatorname{erf}(x)| \;\leq\; 2^{-s}\,|\operatorname{erf}(x)| \;\leq\; 2^{-t'-1}\,2^{-E-1}\,|\operatorname{erf}(x)|.$$

Puisque $x < 0$, $\operatorname{erfc}(x) \geq 1$. De plus, $|\operatorname{erf}(x)| = \operatorname{erf}(|x|) \leq 2|x| \leq 2^{E+1}$, d'où le résultat.

**Troisième cas.**
Même raisonnement que pour le second cas, mais en utilisant $\operatorname{erfc}(x) \geq 1/8$.

**Quatrième cas.**
Ici, $x \geq 1$ et $s \geq t' + 3 + E + x^2 \log_2(e)$. De là

$$|R - \operatorname{erf}(x)| \;\leq\; 2^{-s}\,\operatorname{erf}(x) \;\leq\; 2^{-t'-1}\,2^{-E-2}\,e^{-x^2}\,|\operatorname{erf}(x)|.$$

Puisque $x \geq 1$, $\operatorname{erf}(x) \leq 1$ et $\operatorname{erfc}(x) \geq e^{-x^2}/(4x) \geq e^{-x^2}/(4 \cdot 2^E)$. D'où le résultat annoncé. □

Comme conséquence du lemme, on a

$$|(1 - R) - \operatorname{erfc}(x)| \leq 2^{-t'-1}\,\operatorname{erfc}(x).$$

De là, $|(1 - R)| \leq 2\operatorname{erfc}(x)$. Or, puisque $(1 \ominus R) = (1 - R)\,\langle 1 \rangle$,

$$|(1 \ominus R) - (1 - R)| \;\leq\; |(1 - R)|\,2^{1-(t'+3)} \;\leq\; 2^{-t'-1}\,\operatorname{erfc}(x).$$

Enfin $|(1 \ominus R) - \operatorname{erfc}(x)| \leq |(1 \ominus R) - (1 - R)| + |(1 - R) - \operatorname{erfc}(x)| \leq 2^{-t'}\operatorname{erfc}(x)$ ce qui prouve que $(1 \ominus R)$ est une approximation de $\operatorname{erfc}(x)$ avec une erreur relative bornée par $2^{-t'}$.

### 1.5.6 Implémentation de l'équation (1.3) en pratique

Dans cette section, nous montrons comment utiliser l'équation (1.3) pour calculer une valeur approchée de $\operatorname{erfc}(x)$ (nous supposons à nouveau, contrairement à la section précédente, que $x > 0$) :

$$\operatorname{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}}\left(1 + \sum_{n=1}^{N-1}(-1)^n\,\frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{(2x^2)^n}\right) + \varepsilon_N^{(3)}(x)$$

où $\varepsilon_N^{(3)}(x)$ est le reste, borné par l'inégalité (1.4).

La particularité de cette formule vient du fait que le reste ne peut pas être rendu arbitrairement petit. En fait, étant donné $x$, la borne (1.4) commence par décroître en fonction de $N$, jusqu'à atteindre une valeur optimale pour $N = \lfloor x^2 + 1/2 \rfloor$. Pour des valeurs de $N$ plus grandes, la borne (1.4) est croissante. Ainsi, étant donné une erreur cible $2^{-t'}$, il se peut qu'aucune valeur de $N$ ne convienne. Dans ce cas, l'équation (1.3) ne peut pas être utilisée. En particulier, cette équation n'est jamais utilisable lorsque $0 < x < 1$. Jusqu'à la fin de cette section, nous supposerons donc, sans perte de généralité, que $x \geq 1$.

Réciproquement, lorsque l'erreur cible peut être réalisée, on peut choisir n'importe quelle valeur de $N$ entre deux valeurs $N_{\min}$ et $N_{\max}$. Évidemment, nous sommes intéressés par la plus petite des deux, $N_{\min}$. Pour trouver une surestimation de $N_{\min}$ nous utiliserons le lemme 1.10.

Nous donnons simplement ici les résultats principaux nécessaires à l'implémentation. Les techniques pour les démontrer sont exactement les mêmes que celles utilisées pour les équations (1.1) et (1.2).

**Lemme 1.19.** *L'inégalité suivante est vérifiée dès que $x \geq 1$ :*

$$|\varepsilon_N^{(3)}(x)| \leq \frac{e^{-x^2}}{x}\left(\frac{N}{ex^2}\right)^N.$$

*Démonstration.* Le résultat s'obtient à partir de la borne donnée par l'équation (1.4). Le produit $1 \cdot 3 \cdots (2N-1)$ vaut $(2N)!/(2^N N!)$ et on utilise ensuite les bornes données au lemme 1.8. □

**Proposition 1.20.** *Si $N$ vérifie*

$$\frac{N}{ex^2}\log_2\left(\frac{N}{ex^2}\right) \leq \frac{-t'-3}{ex^2},$$

*le reste $\varepsilon_N^{(3)}(x)$ est majoré par* $\operatorname{erfc}(x)\, 2^{-t'-1}$.

*Démonstration.* On utilise le lemme précédent et l'inégalité $\operatorname{erfc}(x) \geq e^{-x^2}/(4x)$ donnée par le lemme 1.11. □

Cette proposition et le lemme 1.10, vont nous servir à déterminer une surestimation $N$ de $N_{\min}$. Cependant, puisque justement il s'agit d'une surestimation, il est possible que la valeur $N$ que nous allons calculer soit également supérieure à $N_{\max}$. Pour parer à cette possibilité, remarquons deux choses : d'une part, il n'est jamais nécessaire de prendre $N$ plus grand que $x^2$ (puisque nous savons que le reste $\varepsilon_N^{(3)}$ est minimal pour $N = \lfloor x^2 - 1/2 \rfloor$), et d'autre part, une valeur de $N$ étant donnée, il est toujours possible de voir si l'inéquation de la proposition 1.20 est vérifiée ou non.

La règle pour le calcul de $N$ est alors la suivante (on calcule $a = (-t'-3)/ex^2$ en choisissant les modes d'arrondi de façon à obtenir une sous-estimation $a_d$ de $a$) :

- si $a_d < -\log_2(e)/e$, l'équation (1.3) ne peut pas être utilisée ;
- si $a_d \in [-\log_2(e)/e,\, 0[$, poser $N_0 \geq (ex^2) \cdot a_d / \log_2(-a_d)$ et appliquer le test suivant :
  - si $N_0 \leq x^2$, poser $N = N_0$,
  - sinon poser $N \simeq x^2$ et tester si $\dfrac{N}{ex^2}\log_2\left(\dfrac{N}{ex^2}\right) \leq a_d$,
  - si cette relation n'est pas vérifiée, l'équation (1.3) ne peut pas être utilisée.

Bien sûr, en pratique, $-\log_2(e)/e$ est remplacé par une valeur approchée par excès. De même, pour le test $N_0 \leq x^2$, la valeur $x^2$ est remplacée par une valeur approchée par défaut.

Une fois que la valeur de $N$ est fixée, la somme

$$S(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} + \sum_{n=1}^{N-1}(-1)^n \frac{e^{-x^2}}{x\sqrt{\pi}} \cdot \frac{1 \cdot 3 \cdots (2n-1)}{(2x^2)^n}$$

est calculée en utilisant l'algorithme 1.1 avec les paramètres $y = 1/(2x^2)$, $\alpha_0 = e^{-x^2}/(x\sqrt{\pi})$ et, pour $k \geq 1$, $\alpha_k = -(2k-1)$. En pratique, on utilise $\alpha_k = (2k-1)$ et on ajoute et soustrait successivement $\mathsf{acc}$ à la somme partielle (les variables $\mathsf{acc}$, $\mathsf{tmp}$, $i$, $k$, etc. font toujours référence aux variables de l'algorithme 1.1 décrit en page 42).

Lors du calcul de $\alpha_0$, les modes d'arrondi sont choisis de telle sorte que la valeur calculée surestime la valeur exacte. De plus, lorsque acc est mis à jour, l'arrondi est effectué par excès. Ainsi, la boucle principale peut être arrêtée dès que

$$k = N \quad \text{ou} \quad \mathsf{acc} \cdot 2^{Fi} < 2^{-t'-1}\, e^{-x^2}/(4x)$$

où $F$ est l'exposant de $y$. L'algorithme est résumé en figure « algorithme 1.4 ».

Les erreurs d'arrondi sont bornées à l'aide de la proposition suivante :

**Proposition 1.21.** *Si l'algorithme 1.4 est utilisé pour calculer une valeur approchée $\widehat{S(x)}$ de la somme $S(x)$, on a :*

$$\widehat{S(x)} = \frac{e^{-x^2}}{x\sqrt{\pi}}\,\langle 16N\rangle + \sum_{n=1}^{N-1}(-1)^n\,\frac{e^{-x^2}}{x\sqrt{\pi}}\cdot\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n}\,\langle 16N\rangle\,.$$

*De là,*

$$\left|\widehat{S(x)} - S(x)\right| \le \gamma_{16N}\,\frac{e^{-x^2}}{x\sqrt{\pi}}\left(1 + \sum_{n=1}^{N-1}\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n}\right) \le \gamma_{16N}\,\frac{e^{-x^2}}{x\sqrt{\pi}}\cdot\frac{3}{2}.$$

*Démonstration.* Dans l'algorithme, $\widehat{y} = y\,\langle 2\rangle$, et l'exponentiation binaire conduit donc à

$$\widehat{z} = z\,\langle 3L-1\rangle\,.$$

La borne finale est alors

$$\widehat{S(x)} = \frac{e^{-x^2}}{x\sqrt{\pi}}\,\langle 8N+3\rangle + \sum_{n=1}^{N-1}(-1)^n\,\frac{e^{-x^2}}{x\sqrt{\pi}}\cdot\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n}\,\langle 8N+3\rangle\,.$$

On majore $\langle 8N+3\rangle$ par $\langle 16N\rangle$.

Nous prouvons à présent que

$$1 + \sum_{n=1}^{N-1}\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n} \le \frac{3}{2} \quad \text{pour } N \le \lfloor x^2 + 1/2\rfloor.$$

Puisque $N \le \lfloor x^2+1/2\rfloor$, $N \le x^2+1$. De plus, le terme général est décroissant, donc on peut écrire

$$\sum_{n=1}^{N-1}\frac{1\cdot 3\cdots(2n-1)}{(2x^2)^n} \;\le\; (N-1)\frac{1}{2x^2} \;\le\; \frac{1}{2}.$$

□

En utilisant cette proposition, on obtient la règle suivante pour choisir la précision de travail $t$ :

$$\boxed{t \ge t' + 6 + \lceil \log_2(N)\rceil\,.}$$

### 1.5.7 Implémentation de la fonction erf à l'aide de l'équation (1.3)

Nous terminons notre étude en expliquant rapidement comment l'équation (1.3) peut être utilisée pour calculer $\operatorname{erfc}(x)$ lorsque $x \le -1$ et pour calculer $\operatorname{erf}(x)$ pour $x \ge 1$ (le cas symétrique $x < -1$ étant identique).

Notons d'abord que $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = 1 + \operatorname{erf}(-x) = 2 - \operatorname{erfc}(-x)$. Ainsi, on obtient $\operatorname{erfc}(x)$ en calculant une valeur approchée $R$ de $\operatorname{erfc}(-x)$ avec une erreur relative plus petite qu'une certaine borne $2^{-s}$ bien choisie et en calculant $2 \ominus R$ en précision $t'+3$.

**Algorithme :** `erfcByEquation3`

**Input** : un nombre flottant x,
la précision de travail t,
la précision cible t',
$\mathsf{L} \in \mathbb{N}^\star$, $\mathsf{N} \in \mathbb{N}^\star$.

**Output** : une valeur approchée de erfc($x$) dont l'erreur relative est inférieure à $2^{-t'-1}$ obtenue en utilisant l'équation (1.3)

`/* toutes les opérations sont effectuées en précision` $t$ `*/`

$\mathsf{E} \leftarrow$ exponent(x);
$\mathsf{G} \leftarrow \lceil x * x * \log_2(e) \rceil$ ; // arrondi par excès.
acc $\leftarrow$ x $*$ x ; // opération réalisée en précision t $+ 2$ E, arrondi par défaut
y $= 2 *$ acc;
y $\leftarrow 1/$y ; // arrondi par excès
acc $\leftarrow$ exp($-$acc) ; // arrondi par excès
tmp $\leftarrow$ x $* \sqrt{\pi}$ ; // arrondi par défaut
acc $\leftarrow$ acc/tmp ; // arrondi par excès
F $\leftarrow$ exponent(y) ;
z $\leftarrow$ power(y, L) ; // arrondi par excès
S $\leftarrow [0, \ldots, 0]$ ;
i $\leftarrow 0$ ;
k $\leftarrow 0$ ;
**repeat**
  **if** (k mod 2) $= 0$ **then** S[i] $\leftarrow$ S[i] $+$ acc **else** S[i] $\leftarrow$ S[i] $-$ acc ;
  k $\leftarrow$ k $+ 1$;
  **if** i $=$ L $- 1$ **then**
    i $\leftarrow 0$ ;
    acc $\leftarrow$ acc $*$ z ; // arrondi par excès
  **else**
    i $\leftarrow$ i $+ 1$ ;
  **end**
  acc $\leftarrow$ acc $* (2 *$ k $- 1)$ ; // arrondi par excès
**until** k $=$ N **or** exponent(acc) $< -$t' $- 3 -$ F $*$ i $- G - E$ ;

`/* À présent` $S(y)$ `est calculé à partir des` $S_i$ `par le schéma de Horner */`

R $\leftarrow$ S[L $- 1$] ;
**for** i $\leftarrow$ L $- 2$ **downto** $0$ **do**
  R $\leftarrow$ S[i] $+$ y $*$ R ;
**end**
**return** R;

**Algorithme 1.4 :** Évaluation de la fonction erfc par l'équation (1.3)

De la même façon, puisque $\mathrm{erf}(x) = 1 - \mathrm{erfc}(x)$, on obtient $\mathrm{erf}(x)$ en calculant une valeur approchée $R$ de $\mathrm{erfc}(x)$ avec une erreur relative inférieure à une borne $2^{-s}$ bien choisie, puis en calculant $1 \ominus R$ en précision $t' + 3$.

Les valeurs appropriées de $s$ sont données par les deux lemmes suivants (les démonstrations sont laissées au lecteur) :

**Lemme 1.22.** *On a* $|R - \mathrm{erfc}(x)| \leq 2^{-t'-1}\,\mathrm{erfc}(-x)$ *dès lors que* $x \geq 1$ *et que* $s$ *est choisi conformément à la règle suivante :*

$$\boxed{s \geq t' + 2 - E - x^2 \log_2(e).}$$

Remarquons que lorsque $t' + 2 - E - x^2 \log_2(e) \leq 1$, il n'y a même pas lieu de calculer une valeur approchée de $\mathrm{erfc}(x)$ et d'effectuer la soustraction : la valeur 2 peut être directement retournée comme résultat. En effet, $t' + 2 - E - x^2 \log_2(e) \leq 1$ implique que $e^{-x^2}/x \leq 2^{-t'}$ d'où

$$\mathrm{erfc}(x) \leq 2^{-t'} \leq 2^{-t'}\,\mathrm{erfc}(-x).$$

Puisque $\mathrm{erfc}(x) = 2 - \mathrm{erfc}(-x)$, cela signifie que 2 est une valeur approchée de $\mathrm{erfc}(-x)$ avec une erreur relative inférieure à $2^{-t'}$.

**Lemme 1.23.** *On a* $|R - \mathrm{erfc}(x)| \leq 2^{-t'-1}\,\mathrm{erf}(x)$ *dès lors que* $x \geq 1$ *et que* $s$ *est choisi conformément à la règle suivante :*

$$\boxed{s \geq t' + 3 - E - x^2 \log_2(e).}$$

La même remarque est encore valable dans ce cas : si $t' + 3 - E - x^2 \log_2(e) \leq 1$, la valeur 1 peut directement être retournée en tant que valeur approchée de $\mathrm{erf}(x)$ avec une erreur relative majorée par $2^{-t'}$.

## 1.6 Résultats expérimentaux

Nous avons donné tous les détails nécessaires à l'implémentation des trois équations (1.1), (1.2), et (1.3). Chacune peut être utilisée pour obtenir une valeur approchée de $\mathrm{erf}(x)$ ou $\mathrm{erfc}(x)$. Nous avons aussi donné des estimations de l'ordre de troncature $N$ et de la précision de travail $t$. Dans tous les cas, $\mathcal{O}(\sqrt{N})$ multiplications en précision $t$ et $\mathcal{O}(N)$ additions et multiplications/divisions par des petits entiers sont nécessaires pour calculer la somme. Par suite, et pour chaque équation, la complexité binaire de l'algorithme est $\mathcal{O}(tN + M(t)\sqrt{N})$ où $M(t)$ désigne la complexité binaire du produit de deux nombres flottants en précision $t$.

Il faut bien voir que des réalités différentes se cachent derrière cette formule de complexité. En effet, les entrées de l'algorithme sont le point $x$ et la précision cible $t'$. Les valeurs $N$ et $t$ sont donc des fonctions de $x$ et $t'$. Or, comme on a pu le voir dans les sections précédentes, ces fonctions sont très différentes suivant l'équation utilisée.

### 1.6.1 Comment choisir la meilleure équation

Évidemment, nous aimerions choisir automatiquement l'équation à utiliser de façon à minimiser le temps de calcul, en fonction de l'entrée $(x, t')$. Pour cela, nous devons donc comparer les complexités précises des trois équations. Or il semble bien difficile de faire cette étude théoriquement :

— les estimations de $N$ et $t$ sont différentes pour chacune des trois équations. Elles dépendent de $x$ et $t'$ de façon complexe et intriquée. De plus, même pour une équation donnée, ces estimations sont définies par morceaux et il est donc nécessaire d'étudier beaucoup de cas différents ;

— de plus, comparer les trois méthodes nécessite de se donner des hypothèses précises sur l'implémentation de l'arithmétique flottante sous-jacente. Par exemple, quelle est la complexité $M(t)$ d'une multiplication, quelle est la complexité d'une évaluation de exp ?

En règle générale, la multiplication de deux nombres flottants en précision $t$ utilise des algorithmes différents suivant que $t$ est grand ou pas : on se reportera à la section 2.4 de [FHL+07] pour un aperçu de ce qui est fait dans la bibliothèque `MPFR`. Trois algorithmes différents sont utilisés dans `MPFR`. Chacun de ces algorithmes utilise, comme brique de base, la multiplication entre deux entiers de tailles arbitraires.`MPFR` s'appuie sur la bibliothèque `GMP` pour effectuer ces multiplications entre grands entiers. Or `GMP` utilise quatre algorithmes différents suivant la taille des entrées. La complexité effective $M(t)$ est donc très difficile à estimer finement à moins que $t$ soit très grand (auquel cas la complexité asymptotique est atteinte).

Dans `MPFR`, l'évaluation de $\exp(x)$ est effectuée en utilisant trois algorithmes différents suivant la précision $t$ demandée (voir la fin de la section 2.5 de [FHL+07] pour un bref aperçu) :

— une évaluation naïve de la série de Taylor est utilisée lorsque $t$ est plus petit qu'un certain paramètre $t_0$ ;

— la technique de groupage de Smith est utilisée lorsque $t$ se situe entre $t_0$ et un second seuil $t_1$ ;

— enfin, si $t \geq t_1$, une méthode de scindage binaire est utilisée [Jea00, Bre76].

Les valeurs $t_0$ et $t_1$ ont été optimisées (et sont donc différentes) pour chaque architecture.

Ainsi la question du choix optimal d'une équation parmi les trois proposées dans ce chapitre est bien plus une affaire de pratique que de considérations théoriques. Nous avons implémenté les trois algorithmes en C en utilisant `MPFR` pour l'arithmétique flottante sous-jacente. Nous sommes actuellement en discussion avec les développeurs de `MPFR` pour y intégrer notre implantation.

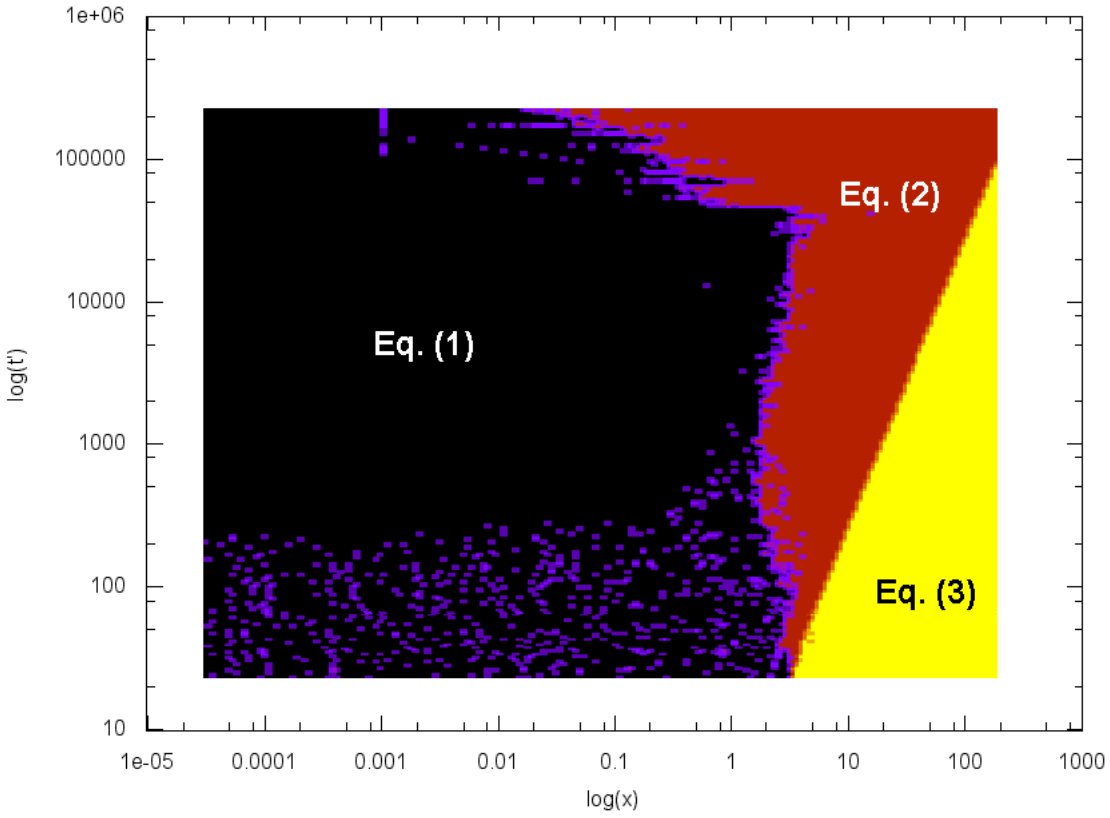



**Figure 1.7 :** Comparaison des temps d'exécution des trois implémentations de erf

Afin de mettre en évidence expérimentalement quelle équation est la plus efficace en fonction

du couple $(x, t')$, nous avons exécuté les trois implémentations sur un grand échantillon de valeurs de $x$ et de $t'$. Pour chaque couple, nous avons comparé les temps d'exécution de chacune des implémentations lors du calcul de $\mathrm{erf}(x)$. Les résultats expérimentaux sont consignés en figure 1.7. Les couleurs indiquent quelle implémentation est la plus rapide. Les tests ont été effectués sur un processeur Intel Pentium D 32-bits cadencé à 3.00 GHz et avec 2.00 Go de mémoire vive, tournant sous Linux 2.6.26 avec `MPFR` 2.3.1 et gcc 4.3.3. Les seuils de `MPFR` pour cette architecture sont $t_0 = 528$ et $t_1 = 47\,120$.

La frontière entre les équations (1.1) et (1.2) semble présenter trois phases suivant la valeur de $t'$. Ces phases correspondent approximativement aux seuils utilisés par `MPFR` pour l'implémentation de la fonction exp. Puisque l'équation (1.2) nécessite le calcul de $\exp(-x^2)$ alors que l'équation (1.1) n'en a pas besoin, il ne s'agit probablement pas d'une coïncidence. Nous observons tout simplement l'impact du calcul de $\exp(-x^2)$ sur le temps de calcul total.

La frontière entre les équations (1.2) et (1.3) est plus régulière. En fait, on observe expérimentalement que l'équation (1.3) est plus intéressante que les deux autres, dès lors qu'on peut l'utiliser. La figure 1.8 montre les temps d'exécution pour $t' = 632$ en fonction de $x$ : pour les petites valeurs de $x$, l'équation (1.3) ne permet pas d'atteindre la précision requise. Mais pour $x \simeq 15$, elle devient utilisable et est immédiatement cinq fois plus rapide que les deux autres équations. Ainsi, le domaine où l'équation (1.3) est meilleure que les autres est donné par les points pour lesquels l'inégalité de la proposition 1.20 a une solution. Autrement dit, les points pour lesquels

$$\frac{-s-3}{ex^2} \geq \frac{-\log_2(e)}{e},$$

où $s \gtrsim t' + 3 - E - x^2 \log_2(e)$ est la précision intermédiaire donnée par le lemme 1.23. Aussi, la frontière entre les équations (1.2) et (1.3) est approximativement donnée par la loi $\log(t') \simeq 2\log(x)$, ce qui correspond aux observations.

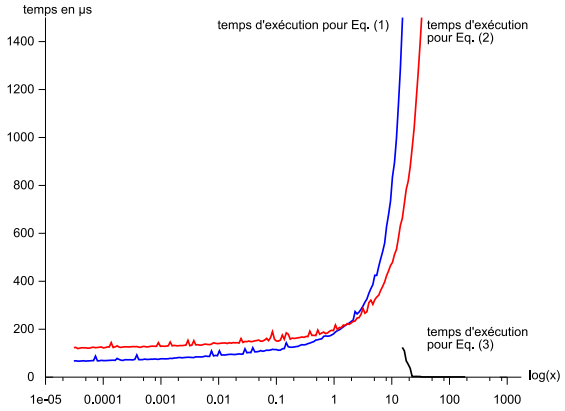



**Figure 1.8 :** Temps d'exécution des trois implémentations, en fonction de $\log(x)$, lorsque $t' = 632$

### 1.6.2 Comparaison avec d'autres implémentations

Nous avons comparé les implémentations des trois équations avec deux autres implémentations de référence : `MPFR` et Maple. `MPFR` donne lieu à la comparaison la plus pertinente puisqu'on compare réellement deux choses comparables : l'arithmétique flottante sous-jacente à l'implémentation est la même pour la fonction erf de `MPFR` et pour notre implémentation. Du coup, la différence dans les temps de calcul entre l'implémentation de `MPFR` et les implémentations proposées ici tient vraiment aux différences algorithmiques.

Maple utilise une arithmétique flottante décimale. De plus, il s'agit d'un langage interprété ; la comparaison entre Maple et notre implémentation est donc moins pertinente. Néanmoins Maple est très largement utilisé et constitue une des rares implémentations de erf et erfc en précision arbitraire.

Les expériences reportées dans la table de la figure 1.9 ont été effectuées sur une machine équipée d'un processeur 32 bits Intel Xeon cadencé à 2.40 GHz avec 2 Go de mémoire vive. Il tournait sous Linux 2.6.22 et les versions utilisées sont `MPFR` 2.3.1, gcc 4.1.2 et Maple 11. Les valeurs de $x$ ont été choisies au hasard avec la même précision $t'$ que la précision cible ; la table n'indique qu'une valeur approchée à chaque fois. La précision cible $t'$ est exprimée en bits. Maple est utilisé avec la variable Digits fixée à $\lfloor t'/\log_2(10) \rfloor$. Cela correspond approximativement à la même précision exprimée en arithmétique décimale. Maple retient les valeurs précédemment calculées ; il est donc impossible de répéter le même calcul plusieurs fois dans le but de mesurer le temps d'une exécution en calculant le temps moyen. Cette difficulté est contournée en évaluant successivement $\mathrm{erf}(x)$, $\mathrm{erf}(x+h)$, $\mathrm{erf}(x+2h)$, etc. où $h$ est une petite valeur d'incrément.

| $x$ | $t'$ | Eq. 1.1 | Eq. 1.2 | Eq. 1.3 | `MPFR` | Maple |
|---|---|---|---|---|---|---|
| 0.000223 | 99 | 29 µs | 47 µs | - | **14 µs** | 283 µs |
| 0.005602 | 99 | 34 µs | 48 µs | - | **17 µs** | 282 µs |
| 0.140716 | 99 | 40 µs | 53 µs | - | **25 µs** | 287 µs |
| 3.534625 | 99 | 96 µs | **84 µs** | - | 125 µs | 382 µs |
| 88.785777 | 99 | 277 520 µs | 6181 µs | **< 1 µs** | 2 µs | 18 µs |
| 0.000223 | 412 | **55 µs** | 87 µs | - | 76 µs | 739 µs |
| 0.005602 | 412 | **62 µs** | 94 µs | - | 104 µs | 783 µs |
| 0.140716 | 412 | **88 µs** | 109 µs | - | 186 µs | 870 µs |
| 3.534625 | 412 | 246 µs | **198 µs** | - | 663 µs | 1 284 µs |
| 88.785777 | 412 | 289 667 µs | 9 300 µs | **< 1 µs** | 2 µs | 21 µs |
| 0.000223 | 1 715 | **311 µs** | 562 µs | - | 1 769 µs | 2 513 µs |
| 0.005602 | 1 715 | **393 µs** | 616 µs | - | 2 490 µs | 2 959 µs |
| 0.140716 | 1 715 | **585 µs** | 748 µs | - | 4 263 µs | 3 968 µs |
| 3.534625 | 1 715 | 1 442 µs | **1 260 µs** | - | 11 571 µs | 8 850 µs |
| 88.785777 | 1 715 | 343 766 µs | 22 680 µs | **< 1 µs** | 2 µs | 42 µs |
| 0.000223 | 7 139 | **3 860 µs** | 7 409 µs | - | 28 643 µs | 38 846 µs |
| 0.005602 | 7 139 | **4 991 µs** | 7 959 µs | - | 40 066 µs | 51 500 µs |
| 0.140716 | 7 139 | **7 053 µs** | 9 227 µs | - | 64 975 µs | 79 308 µs |
| 3.534625 | 7 139 | 14 744 µs | **14 144 µs** | - | 157 201 µs | 191 833 µs |
| 88.785777 | 7 139 | 628 527 µs | 96845 µs | 46 µs | **2 µs** | 213 µs |
| 0.000223 | 29 717 | **63 ms** | 108 ms | - | 654 ms | 1 140 ms |
| 0.005602 | 29 717 | **79 ms** | 119 ms | - | 881 ms | 1 539 ms |
| 0.140716 | 29 717 | **108 ms** | 137 ms | - | 1 375 ms | 2 421 ms |
| 3.534625 | 29 717 | 202 ms | **198 ms** | - | 2 968 ms | 5 320 ms |
| 88.785777 | 29 717 | 2 005 ms | **898 ms** | - | 39 760 ms | 243 690 ms |

**Figure 1.9 :** Temps d'exécution de plusieurs implémentations de erf

Les cas où l'équation (1.3) ne peut pas être utilisée parce que la précision cible est trop élevée sont représentés par le symbole « - ». Notre implémentation est la plus rapide, à l'exception de

quelques rares cas. En faible précision et pour de petites valeurs de $x$, la bibliothèque `MPFR` est la plus rapide parce qu'elle utilise une évaluation directe qui est un peu plus rapide lorsque le rang de troncature est faible.

Le cas $x \simeq 88.785777$ et $t' = 7139$ s'explique autrement. En fait, cette situation correspond à la remarque qui suit le lemme 1.23 page 61 : $t' + 3 - E - x^2 \log_2(e) \leq 1$. Aussi, il n'y a aucun calcul à faire et la valeur 1 peut être retournée immédiatement. Dans notre implémentation, la valeur $t' + 3 - E - x^2 \log_2(e)$ est calculée en utilisant `MPFR` en faible précision. Ce calcul nécessite 46 µs. `MPFR` effectue le même genre de test mais utilise pour cela l'arithmétique flottante présente en matériel (précision `double`). Cela explique que la réponse soit obtenue en 2 µs seulement.

## 1.7 Conclusion et perspectives de ce travail

Nous avons proposé trois algorithmes pour évaluer efficacement les fonctions erf et erfc en précision arbitraire. Ces algorithmes s'appuient sur trois formules de sommation différentes mais qui présentent toutes la même structure générale. Pour évaluer ces sommes, on peut tirer parti de leur structure en utilisant un algorithme dû à Smith qui permet d'effectuer le calcul de la somme avec une complexité binaire $\mathcal{O}(Nt + M(t)\sqrt{N})$ au lieu de $\mathcal{O}(N\,M(t))$ qui correspond à l'approche classique d'une évaluation par schéma de Horner.

Nous avons calculé des formules closes permettant de majorer le rang de troncature $N$ et nous avons complètement étudié les effets des erreurs d'arrondi qui se propagent dans l'évaluation de la somme. Nous en avons déduit des formules closes pour calculer la précision de travail requise.

Par ailleurs, nous avons implémenté les trois algorithmes en C et comparé leur efficacité en pratique. Cette étude a montré que le développement asymptotique est le meilleur choix, dès lors qu'il permet d'obtenir la précision demandée. Lorsque la précision demandée est trop élevée, le développement asymptotique n'est pas assez précis et on doit choisir d'utiliser l'une des deux autres méthodes. Le domaine où il est plus intéressant d'utiliser l'une que l'autre de ces deux équations dépend de l'arithmétique sous-jacente. En pratique, il faut déterminer pour chaque architecture des seuils spécifiques. Nous avons aussi comparé nos implémentations à l'implémentation de erf fournie par `MPFR` et Maple. Notre implémentation est la plus rapide dans pratiquement tous les cas. Lorsque la précision devient grande, nos algorithmes représentent un gain de temps considérable par rapport à Maple ou `MPFR`.

Il faut toutefois faire quelques remarques qui permettraient d'améliorer encore ce travail. D'abord, nous avons fait l'hypothèse qu'aucun dépassement de capacité n'apparaît pendant l'évaluation. Ce choix est légitimé par le fait que la plage d'exposants proposée dans `MPFR` est immense et peut souvent être considérée comme infinie en pratique. Cependant, pour être tout à fait complet, nous devrions en tenir compte afin de garantir la qualité du résultat y compris dans le cas improbable où un tel dépassement de capacité se produirait.

Comme nous l'avons vu (cf. figure 1.8 page 63), l'équation (1.3) est bien plus efficace que les équations (1.1) et (1.2) lorsqu'elle peut être utilisée. On a donc intérêt à essayer d'étendre le domaine d'application de l'équation (1.3). On peut s'y prendre de la façon suivante : considérons un point $x$ pour lequel l'équation (1.3) ne donne pas suffisamment de précision. Supposons qu'on puisse écrire $x = x_0 - h$ où $h$ est assez petit et où $x_0$ est dans le domaine d'application de l'équation (1.3). Alors, une valeur approchée de $\mathrm{erf}(x)$ est obtenue en considérant le développement de Taylor de erf en $x_0$ :

$$\mathrm{erf}(x) = \mathrm{erf}(x_0) + \sum_{i=1}^{+\infty} a_i(-h)^i \quad \text{où} \quad a_i = \frac{\mathrm{erf}^{(i)}(x_0)}{i!}.$$

Puisque $h$ est petit, seuls quelques termes de la somme sont nécessaires pour obtenir une bonne approximation. Les coefficients $a_i$ sont de la forme $p_i(x_0)e^{-x_0^2}$, où $p_i$ est un polynôme de degré $2i-2$

vérifiant une récurrence simple ; les $a_i$ sont donc facilement calculables. La valeur $\mathrm{erf}(x_0)$ est calculée efficacement en utilisant l'équation (1.3) (la valeur $e^{-x_0^2}$ est calculée à ce moment-là). Ceci permet donc de s'appuyer sur l'efficacité de l'équation (1.3) pour évaluer $\mathrm{erf}(x)$ bien que $x$ soit hors du domaine d'application directe de l'équation.

En réalité, la technique que nous venons d'esquisser repose sur deux éléments :

— l'idée d'effectuer le prolongement analytique de erf en un point $x_0$ pour lequel $\mathrm{erf}(x_0)$ est facile à calculer ;

— le fait que les coefficients de la série de Taylor se déduisent aisément de $\mathrm{erf}(x_0)$ et de $e^{-x_0^2}$.

Si la série de Taylor de erf en $x_0$ est si régulière, c'est parce que la fonction erf fait partie d'une grande classe de fonctions appelées *holonomes* ou encore *D-finies*. Une fonction est holonome lorsqu'elle est solution d'une équation différentielle linéaire à coefficients polynomiaux. En l'occurrence, erf est solution de l'équation

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + 2x \frac{\mathrm{d}y}{\mathrm{d}x} = 0.$$

Lorsqu'une fonction holonome est analytique en un point $x_0$, les coefficients du développement de Taylor en $x_0$ vérifient une récurrence. C'est cette propriété qui rend le prolongement analytique intéressant en pratique pour évaluer erf.

D. V. Chudnovsky et G. V. Chudnovksy ont proposé [CC90] un algorithme quasi-linéaire (de complexité binaire $\mathcal{O}(M(t)\,\log(t)^3)$) permettant d'évaluer n'importe quelle fonction holonome avec $t$ bits de précision. Cet algorithme, appelé *bit-burst*, repose sur deux idées :

1. Lorsque $x_0$ est un rationnel $p/q$ où $p$ et $q$ ont une petite taille binaire, il est possible d'évaluer $N$ termes de la série de Taylor d'une fonction holonome très rapidement en appliquant une technique classique appelée *scindage binaire*.
2. Pour un nombre flottant $x$ quelconque, on utilise le prolongement analytique récursivement : on écrit $x = x_0 + h$ où $x_0 = p/q$ avec $p$ et $q$ de petits entiers. Ceci conduit à évaluer $f(x_0)$ (par scindage binaire) ainsi qu'une série en $h$ (dont les coefficients dépendent de $f(x_0)$ et éventuellement des premières dérivées de $f$ en $x_0$). Cette série est elle-même évaluée en décomposant $h$ sous la forme $h = h_0 + h'$, etc.

Brent avait déjà proposé en 1976 un algorithme pour le calcul de la fonction exponentielle exploitant le scindage binaire [Bre76]. Le *bit-burst* est en fait une généralisation de cet algorithme à n'importe quelle fonction holonome. En pratique, la constante dans le « $\mathcal{O}$ » n'est pas négligeable et l'algorithme n'est intéressant que pour des précisions assez grandes. Il s'agit néanmoins d'une piste d'amélioration significative.

Une remarque concerne l'efficacité pratique des méthodes que nous avons exposées, lorsque la précision devient très grande. L'algorithme de sommation de Smith, mais aussi l'algorithme *bit-burst*, nécessitent une certaine quantité de mémoire pour stocker des résultats intermédiaires. Cela pourrait rendre ces techniques inefficaces dans le cas de très grandes précisions : la mémoire vive pourrait alors être insuffisante et l'usage du disque dur comme mémoire secondaire ralentirait considérablement l'exécution. Dans ce cas, il serait sans doute préférable d'utiliser l'approche traditionnelle qui consiste à évaluer toute la somme par le schéma de Horner. Le résultat est alors accumulé à la volée et il n'y a pas besoin de mémoire supplémentaire. Bien que plus lente d'un point de vue théorique, cette méthode pourrait se révéler plus rapide en pratique si elle évite l'usage du disque dur.

Enfin, il est possible d'évaluer les fonctions erf et erfc en utilisant autre chose que des séries. En particulier, ces fonctions possèdent des développements en fractions continues relativement simples, qui pourraient être exploités pour l'évaluation en précision arbitraire. Il s'agit là d'une alternative prometteuse que nous n'avons pas encore étudiée.

# Présentation du logiciel Sollya

*Ce chapitre a pour but de présenter le logiciel Sollya, développé durant la thèse. Nous présentons la philosophie de l'outil, quelques éléments de syntaxe élémentaires ainsi que quelques fonctionnalités clé. Sollya sera le support de nombreux exemples dans la suite du présent manuscrit.*

Dans le prochain chapitre, nous allons nous intéresser au problème de l'évaluation d'une fonction numérique $f$ en précision fixée. Contrairement à ce que nous avons fait pour les fonctions erf et erfc au chapitre précédent, la précision cible ne sera plus un paramètre de l'algorithme mais une donnée connue d'avance. Comme nous le verrons, il est alors possible d'effectuer un certain nombre de calculs préliminaires qui permettent de synthétiser une implémentation parfaitement adaptée à la précision demandée. Cette implémentation sera en général bien plus efficace qu'une implémentation générique du type de celles présentées au chapitre précédent.

Cette problématique de l'approximation en précision fixée constitue une part importante de cette thèse. Le fait de savoir à l'avance quelle est la précision cible ouvre un espace de recherche incroyablement vaste. On peut par exemple décider de découper le domaine en plusieurs sous-domaines où la fonction sera évaluée par des procédés différents. Comment choisir le meilleur découpage ? Va-t-on approcher la fonction par un polynôme ? Une fraction rationnelle ? Un procédé itératif de type Newton-Raphson ? Comment choisir la valeur initiale du procédé itératif ? Comment choisir les coefficients du ou des polynômes/fractions rationnelles ? On peut décider de fixer la valeur de certains coefficients à des valeurs commodes (0, une puissance de 2, la somme de deux puissances de 2). Quel schéma d'évaluation va-t-on choisir pour évaluer les polynômes/fractions rationnelles ?

L'implantation de la fonction numérique passe donc généralement par une importante phase d'expérimentation où l'on explore les nombreuses pistes d'optimisation. Pour que cette phase d'expérimentation porte ses fruits, il faut simplifier au maximum le travail du numéricien en lui offrant un outil intégré au sein duquel il peut rapidement et simplement procéder aux calculs et essais qui guideront ses choix. Au cours de cette thèse, nous avons développé — principalement avec Christoph Lauter — un tel outil : un logiciel appelé *Sollya.*

## Premiers pas avec Sollya

Sollya est un logiciel libre distribué sous licence CeCILL-C et disponible à l'adresse `http://sollya.gforge.inria.fr/`. Il est écrit en langage C et comporte actuellement environ 65 000 lignes de code. Je suis l'auteur d'environ 20 % de ce code.

Sollya se présente sous la forme d'un shell interactif (comme par exemple Maple, Matlab, etc.) dans lequel l'utilisateur peut mener ses expériences. Le logiciel possède une documentation complète [CJJL]. Elle présente une introduction au fonctionnement et à la philosophie générale du logiciel et fournit une description détaillée de chaque commande de Sollya.

Voici, par exemple, un extrait d'une session typique sous Sollya. Dans cet exemple, l'utilisateur définit une fonction $f$, un intervalle `domaine`, calcule une estimation $p$ du polynôme de meilleure approximation de $f$ de degré 5 sur l'intervalle, affiche le polynôme et sa valeur au point $x = \pi$ et $x = 0$, puis finalement, évalue la norme sup [3] de l'erreur $p - f$ sur l'intervalle :

$$\|p - f\|_\infty = \max\{|p(x) - f(x)|,\ x \in [0,\ 4]\}.$$

```
> f = cos(x);
> domaine = [0;4];
> p = remez(f,5,domaine);
> p;
0.9989077212044773996091425051579772155117244175482 5 + x * (1.428568338365498046
9181672543312835592737060191343 6e-2 + x * (-0.5276634312108863451007889090595865
4062238219991645 + x * (1.171931498343972963043596598049536443478257018573 4e-2 +
 x * (4.840874891680153578767092254903391366943967652120 4e-2 + x * (-6.260593102
212291184467705394162111618202742500062e-3)))))
> p(pi);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
-1.0010819785477077015946736054508775113268226524259 8
> p(0);
0.9989077212044773996091425051579772155117244175482 5
> dirtyinfnorm(p-f, domaine);
1.0922787955998166556031368076376432799246391472938 6e-3
```

Nous allons exposer dans ce chapitre les grands principes et fonctionnalités de Sollya. La plupart des algorithmes décrits dans la suite de ce manuscrit ont été implémentés dans le logiciel Sollya et les exemples que nous étudierons seront à peu près tous présentés avec Sollya. Il est donc important de se familiariser avec ce logiciel.

Sollya repose sur un principe simple : toute erreur d'arrondi est signalée. Ainsi, lorsqu'on effectue une opération, on ne risque pas de la croire exacte à tort. Ceci conduit le logiciel à afficher très souvent des messages mettant l'utilisateur en garde contre le fait que le calcul qu'il vient d'effectuer a donné lieu à des erreurs d'arrondi. Ainsi dans l'exemple précédent, l'évaluation de $p$ au point $x = \pi$ n'est pas exacte ; la valeur affichée est donc différente de la vraie valeur $p(\pi)$. C'est explicitement signalé par le message d'avertissement. Lorsque Sollya effectue un calcul et qu'aucune erreur d'arrondi ne se produit, la valeur s'affiche sans avertissement puisqu'elle est correcte. C'est le cas de $p(0)$ puisque $p(0)$ est égal au terme constant de $p$, lequel est exactement représenté en mémoire.

En concordance avec ce principe, le nom des commandes ne laisse en général pas d'ambiguïté sur la nature certifiée ou non du résultat : dans l'exemple, la commande `dirtyinfnorm` calcule une estimation de la norme sup, sans aucune garantie sur la fiabilité du résultat. Il s'agit purement et simplement d'un algorithme numérique, donnant la plupart du temps une estimation de très bonne qualité, mais sans certitude. À l'inverse, Sollya fournit une commande `infnorm` qui calcule un encadrement prouvé de la norme sup ; ce sujet fera l'objet du chapitre 3.

Durant le développement de Sollya, un effort constant a été apporté pour rendre les problèmes arithmétiques invisibles à l'utilisateur. Dans cette optique, Sollya utilise la bibliothèque `MPFR` pour

3. La norme sup (norme du *supremum*) est aussi appelée norme de la convergence uniforme ou norme infini. Dans l'expression *norme infini*, le mot *infini* n'est pas un adjectif : c'est le nom de la norme. En anglais, par exemple, on parle de *infinity norm*. Bien que ce soit un usage courant en français, il est abusif d'accorder *infini* dans cette expression.

permettre à l'utilisateur de fixer la précision (exprimée en bits) avec laquelle il veut travailler. Cette précision est commandée par la variable `prec`, qui vaut 165 par défaut, offrant ainsi d'emblée un cadre confortable.

Mais travailler avec une précision de 165 bits, même si cela offre une certaine sécurité, ne garantit pas qu'un calcul particulièrement mal conditionné ne produira pas un résultat complètement faux. Par exemple, lorsqu'on développe du code pour implémenter une fonction numérique $f$, on manipule souvent des polynômes d'approximation $p$ fournissant plus de 100 bits corrects ; et on veut pouvoir manipuler la fonction $(p-f)/f$ qui représente l'erreur d'approximation sans se soucier de problèmes de précision.

L'utilisateur de Sollya n'a pas à se préoccuper de ce genre de détails car le logiciel essaie toujours d'adapter automatiquement la précision des calculs pour que le résultat soit un arrondi fidèle de la valeur exacte à la précision `prec` (cf. définition 1.1 page 35).

Sollya ne peut malheureusement pas atteindre systématiquement cet objectif. Si le calcul est *vraiment* très mal conditionné, il se pourrait que Sollya ne parvienne pas à trouver une précision de calcul suffisamment élevée pour avoir un résultat digne de confiance. Cette situation est évidemment peu probable et nous ne l'avons jamais observée en pratique.

Il y a un autre cas où Sollya ne peut pas garantir l'arrondi fidèle : lorsque le résultat d'un calcul numérique est exactement 0. Pour comprendre ce phénomène, considérons un calcul dont le résultat exact est un certain réel $x$ ; en pratique, le calcul (effectué avec une certaine précision $t$) donne lieu à des erreurs d'arrondi et on peut seulement affirmer que la valeur exacte est comprise dans un intervalle $[x-\varepsilon,\, x+\varepsilon]$, avec typiquement $\varepsilon = \mathcal{O}(2^{-t})$. On souhaite obtenir un arrondi fidèle de $x$ en précision `prec`. En augmentant la précision de travail $t$, on peut affiner l'intervalle autant qu'on le souhaite. Si $x$ est non nul et que la précision $t$ est suffisamment grande, l'intervalle finit par contenir au plus un nombre flottant de précision `prec` car la grille flottante autour de $x$ est discrète. Ce flottant est alors un arrondi fidèle de $x$ en précision `prec`. Mais 0 est un point d'accumulation de la grille flottante de sorte que, lorsque $x = 0$, on ne parvient jamais à isoler 0 comme unique nombre flottant dans l'intervalle $[-\varepsilon,\, +\varepsilon]$. Ce phénomène est illustré en figure B.

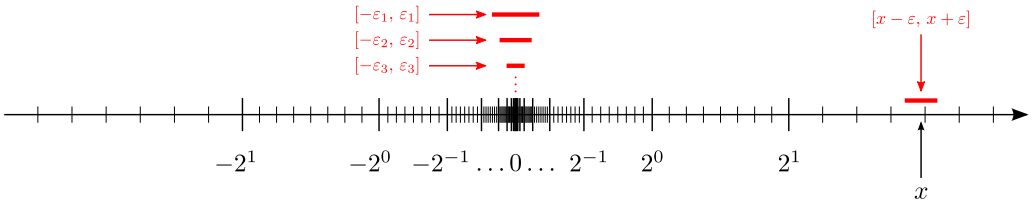



**Figure B :** Si $x \neq 0$, on peut toujours trouver $\varepsilon$ suffisamment petit, tel que $[x-\varepsilon,\, x+\varepsilon]$ ne contienne qu'un seul nombre flottant. Si $x = 0$, aucune valeur de $\varepsilon$ ne convient ($\varepsilon_1$, $\varepsilon_2$, $\varepsilon_3$, ...).

Dans ce cas, Sollya ne peut pas fournir l'arrondi fidèle. Mais il prévient alors explicitement l'utilisateur.

```
> sin(5);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
-0.95892427466313846889315440615599397335246154396459
> sinh(1) - (exp(1)-exp(-1))/2;
Warning: rounding may have happened.
If there is rounding, the displayed value is *NOT* guaranteed to be a faithful r
ounding of the true result.
0
```

Cette idée d'augmenter automatiquement la précision des calculs de façon à garantir la qualité du résultat rend le travail avec Sollya très confortable pour l'utilisateur. Il peut se concentrer sur ce qu'il veut vraiment faire et travailler avec les nombres comme s'il s'agissait de vrais nombres réels. La variable `prec` n'indique pas la précision avec laquelle on souhaite *faire* les calculs, mais bien la précision avec laquelle on souhaite *connaître* le résultat. De ce point de vue, Sollya s'inscrit dans la même démarche que la bibliothèque `iRRAM` de N. Th. Müller [Mü00] par exemple.

Pour calculer un encadrement certifié de la valeur exacte d'une expression numérique, Sollya utilise l'arithmétique d'intervalle.

## L'arithmétique d'intervalle

L'arithmétique d'intervalle consiste à manipuler les intervalles fermés comme s'il s'agissait de nombres réels. Pour une présentation formelle et détaillée de ce sujet, on pourra se reporter aux livres de Moore [Moo79] ou de Neumaier [Neu90].

L'idée générale est la suivante : si on sait qu'un nombre $x$ est compris dans un intervalle $[a, b]$ et qu'un nombre $y$ est compris dans un intervalle $[c, d]$ alors, on sait avec certitude que $x + y$ est compris dans l'intervalle $[a + c, b + d]$, et *a fortiori* dans n'importe quel intervalle l'englobant.

L'arithmétique d'intervalle exploite ce principe en le généralisant à toutes les fonctions numériques : au lieu de calculer de façon approchée avec une fonction numérique $f$, on utilise une fonction $\widetilde{f}$ qui prend en entrée un intervalle $I$ et renvoie en sortie un intervalle $J$ qui vérifie $f(I) \subseteq J$. Ici $f(I)$ désigne l'image (exacte) de $f$ sur l'intervalle $I$ :

$$f(I) = \{y \in \mathbb{R}, \text{ tels que } \exists x \in I,\ f(x) = y\}.$$

En pratique, les bornes de l'intervalle $J$ doivent être des nombres flottants. On utilise alors les modes d'arrondi pour garantir que $J$ est bien un sur-intervalle de l'intervalle exact. Dans l'exemple de l'addition, on prendra donc en fait $J = [\nabla(a + c), \Delta(b + d)]$ où $\nabla$ et $\Delta$ représentent respectivement l'arrondi par défaut et l'arrondi par excès.

### Bibliothèques de fonctions par intervalle

Il existe des bibliothèques logicielles [4] qui fournissent ainsi un certain nombre de fonctions de base (addition, multiplication, mais aussi exp, sin, erf, etc.) en arithmétique d'intervalle. Dans Sollya, nous utilisons la bibliothèque `MPFI` [5] qui permet de travailler en précision arbitraire.

Soit $f$ une fonction de base et $I$ un intervalle. Dans `MPFI`, on peut choisir une précision cible arbitraire $t$ : le résultat $J_t$ de l'évaluation de $f$ sur $I$ par `MPFI` est le plus petit intervalle contenant $f(I)$ et dont les bornes soient des nombres flottants en précision $t$. En d'autres termes, si $f(I)$ est exactement égal à $[u, v]$ (où $u$ et $v$ sont des nombres réels), l'intervalle renvoyé par `MPFI` est

$$J_t = [\nabla_t(u), \Delta_t(v)]\,.$$

`MPFI` fournit aussi quelques constantes ($\pi$ par exemple) vues comme des fonctions constantes : en précision $t$ on obtient par exemple, $\pi() \subseteq J_t = [\nabla_t(\pi), \Delta_t(\pi)]$.

Les fonctions de base sont donc arbitrairement exactes : quitte à augmenter la précision $t$, on obtient un intervalle qui englobe $f(I)$ d'aussi près que l'on veut :

$$\bigcap_{t \in \mathbb{N}^\star} J_t = f(I).$$

4. La page http://www.cs.utep.edu/interval-comp/intsoft.html en recense un grand nombre.
5. distribuée sous licence LGPL et disponible à l'adresse http://gforge.inria.fr/projects/mpfi/

### Composition de fonctions par intervalle

La grande force de l'arithmétique d'intervalle vient du fait que la propriété d'inclusion se conserve lorsqu'on compose les fonctions. Ainsi, il est possible d'évaluer par intervalle n'importe quelle fonction qui s'exprime comme composée des fonctions de base. Par exemple, si $f$ et $g$ sont deux fonctions de base, notons $J$ l'intervalle obtenu en évaluant $g$ par intervalle sur $I$, et $K$ celui obtenu en évaluant $f$ sur $J$. Alors $g(I) \subseteq J$ donc $(f \circ g)(I) \subseteq f(J) \subseteq K$.

Plus généralement, si une fonction $h$ est donnée par une expression formée de compositions de fonctions de base, de constantes et de variables, on peut évaluer $h$ par intervalle en composant simplement les fonctions de base et les constantes de l'arithmétique d'intervalle.

Ce procédé a toutefois des limites : contrairement à ce qu'on avait pour les fonctions de base, l'intervalle $K$ finalement obtenu n'est plus nécessairement arbitrairement précis. Même si l'on augmente la précision $t$ (et même si les calculs étaient faits dans une arithmétique réelle exacte), on peut avoir une inclusion stricte $h(I) \subset K$. Ce phénomène de surestimation est inévitable et vient du fait que les différentes occurrences d'une même variable sont traitées indépendamment les unes des autres. On perd donc l'information de corrélation qu'il y a entre différentes occurrences d'une même variable.

Prenons un exemple. On cherche à obtenir un encadrement de $x(1-x)$ sur l'intervalle $I = [0,\, 1]$. On a $1 - x \subseteq [0,\, 1]$ et donc

$$x(1-x) \subseteq \underbrace{[0,\, 1]}_{I_1} \times \underbrace{[0,\, 1]}_{I_2} \subseteq [0,\, 1]\,.$$

Cependant, l'intervalle image exact est $[0,\, 1/4]$. Le problème vient de la multiplication de $I_1$ par $I_2$ : durant cette multiplication, les intervalles sont traités comme s'ils étaient indépendants. Mais en réalité, si $x$ est proche de 0, $(1 - x)$ est proche de 1 et réciproquement. Cette corrélation entre les intervalles $I_1$ et $I_2$ est ignorée par l'arithmétique d'intervalle. Tout se passe comme si les deux occurrences de $x$ dans l'expression $x(1 - x)$ étaient deux variables distinctes : l'expression $x(1 - y)$ évaluée sur les intervalles $x = [0,\, 1]$ et $y = [0,\, 1]$ donne lieu exactement au même calcul.

L'arithmétique d'intervalle sur des fonctions composées donne donc bien un encadrement de l'image exacte, mais en général cet encadrement est assez large. Ce phénomène est d'autant plus prononcé que l'intervalle $I$ est grand.

Calculer un encadrement précis de l'image de $h(I)$ est un problème difficile : cela revient à calculer de façon certifiée le minimum et le maximum de $h$ sur $I$. Un problème équivalent est de donner un encadrement précis de la norme sup de $h$ sur $I$. Cette question sera étudiée plus en détail au chapitre 3.

### Cas de l'évaluation en un point

Un cas particulier mérite qu'on s'y attarde un peu : celui où l'intervalle $I$ ne contient qu'un seul point : $I = [a,\, a]$. Dans ce cas, il n'y a théoriquement pas de décorrélation : si les calculs étaient effectués en arithmétique réelle exacte, l'intervalle $K$ finalement obtenu serait exactement

$$K = [h(a),\, h(a)]\,.$$

Cependant, les calculs sont effectués en arithmétique flottante inexacte. Imaginons par exemple que $h = (f \circ g)$ où $g$ est une fonction de base :

- la première chose consiste à évaluer $g$ par arithmétique d'intervalle. En général, $g(a)$ n'est pas exactement représentable et on obtient donc un intervalle $J = [\nabla(g(a)),\, \Delta(g(a))]$ de taille strictement positive ;
- ensuite, on évalue l'expression $f$ sur l'intervalle $J$. Celui-ci n'étant plus de taille nulle, des phénomènes de décorrélation peuvent apparaître.

Cependant, cette décorrélation reste faible puisque $J$ est très petit. En outre, si l'on augmente la précision $t$ des calculs, l'intervalle $J$ se rapproche arbitrairement de l'intervalle-point $[g(a), g(a)]$. On peut donc limiter la décorrélation autant qu'on le veut. Ainsi, en augmentant la précision, on peut généralement obtenir un encadrement arbitrairement fin de la valeur exacte de $h(a)$. Ce résultat est vrai pour une grande classe d'expressions $h$ dites *continues Lipschitz* (corollaire 2.1.6 de [Neu90]). C'est cette importante propriété qui est utilisée dans Sollya pour fournir l'arrondi fidèle aussi souvent que possible.

## Autres fonctionnalités de Sollya

L'arithmétique d'intervalle présente au cœur de Sollya est également accessible directement à l'utilisateur de façon transparente. L'exemple suivant illustre le polymorphisme naturel de Sollya : toutes les fonctions de base sont capables de manipuler aussi bien des nombres que des intervalles. La notation `[a]` est un raccourci pour parler de l'intervalle $[a, a]$ constitué uniquement du point $a$. Sollya fournit un mode spécial `midpointmode` qui permet d'afficher les intervalles d'une façon plus intelligible : si l'intervalle $I$ est petit (ce qui est le cas lorsqu'il représente un encadrement fin d'une valeur exacte), sa borne inférieure et sa borne supérieure ont les mêmes premiers chiffres décimaux. On peut afficher les chiffres décimaux communs, suivis d'une indication sur la valeur du premier chiffre à partir duquel les bornes diffèrent. On voit ainsi d'un simple coup d'œil la qualité de l'encadrement.

```
> I=[0;1];
> I*(1-I);
[0;1]
> f=exp(x)*(1-x);
> f([1/2]);
[0.824360635350064073424325393907081785826888050355 07;0.824360635350064073424325
393907081785826888050355 09]
> midpointmode=on;
Midpoint mode has been activated.
> f([1/2]);
0.824360635350064073424325393907081785826888050355~0/1~
```

Sollya n'est pas uniquement une sorte de calculette évoluée capable de travailler en précision arbitraire. La plupart des algorithmes que nous allons décrire dans la suite de ce manuscrit sont implantés dans Sollya. Il permet, entre autres choses, de tracer le graphe d'une fonction numérique sur un intervalle, de trouver les zéros d'une fonction sur un intervalle, de calculer sa norme sup, de calculer des polynômes de meilleure approximation, etc. Un petit langage de script est disponible, qui permet de faire des tests, des boucles, de grouper des éléments au sein de listes, etc. Pour automatiser certaines tâches, l'utilisateur peut aussi écrire ses propres procédures. L'exemple suivant montre comment on peut, en quelques lignes, écrire un petit script pour mesurer la qualité d'approximation d'un polynôme en utilisant le théorème de La Vallée Poussin. La commande `remez`, le théorème de La Vallée Poussin, etc. seront expliqués en détail dans le chapitre suivant.

```
> verbosity = 0!;
> midpointmode = on!;
> qualiteDemandee = 1e-15;
> I = [0;1];
> f = exp(x);
> p = remez(f, 5, I, 1, qualiteDemandee);
>
> L = dirtyfindzeros(diff(p-f),I);
> ptsCritiques = inf(I).:L:.sup(I);
> minimum = @Inf@;
> maximum = 0;
>
> for pt in ptsCritiques do {
    r = evaluate(abs(p-f), [pt]);
    if ( inf(r) < minimum ) then minimum = inf(r);
    if ( sup(r) > maximum ) then maximum = sup(r);
  };
>
> print("L'erreur d'approximation optimale est : ", [minimum,maximum]);
L'erreur d'approximation optimale est :  0.1129569802274786~6/9~e-5
>
> qualiteEffective := (maximum-minimum)/minimum;
> print("Qualite effective :", qualiteEffective);
Qualite effective : 1.689067718616530074715432024636482935781192900552 98e-16
```

Chapitre 2

# Outils pour l'approximation en précision fixée

*Ce chapitre constitue le cœur de la thèse. Nous y étudions diverses questions liées à l'approximation de fonctions en précision fixée. Dans un premier temps, nous présentons les grandes lignes de la démarche utilisée pour implanter une fonction dans une libm. En particulier, nous montrons que l'approximation polynomiale y tient un rôle important. Aussi, nous commençons par rappeler en section 2.2 les résultats et algorithmes classiques d'approximation : approximation polynomiale (théorème d'alternance, théorème de La Vallée Poussin, algorithme de Rémez) et approximation par une combinaison linéaire de fonctions continues quelconques (condition de Haar, algorithme d'échange de Stiefel). Ces résultats ne sont pas nouveaux ; notre contribution en ce domaine vient de l'implantation de ces algorithmes dans le logiciel Sollya.*
*Dans un deuxième temps, en section 2.3, nous étudions la question de l'approximation par des polynômes dont les coefficients sont des nombres représentables en machine. Ce sont ces polynômes qui seront finalement utilisés pour l'implantation pratique de fonctions. Pour répondre à ce besoin, nous proposons une famille d'outils utilisant la programmation linéaire ainsi qu'un algorithme heuristique s'appuyant sur la réduction de réseaux euclidiens.*
*Pour finir le chapitre, nous nous intéressons en section 2.4 au cas de l'approximation par des fractions rationnelles. M. Ercegovac a proposé en 1975 un algorithme (la E-méthode) permettant d'évaluer efficacement une certaine classe de fractions rationnelles en matériel. Nous rappelons le principe et les contraintes de la E-méthode, puis nous montrons comment utiliser les mêmes techniques de programmation linéaire et de réduction des réseaux euclidiens pour fournir des fractions rationnelles à coefficients représentables en machine et qui satisfont aux contraintes de la E-méthode.*

## 2.1 Introduction

Dans le premier chapitre, nous avons montré, à travers l'exemple de la fonction erf, comment il était possible d'implanter efficacement une fonction numérique en précision arbitraire. Dans ce chapitre, nous allons étudier une problématique en apparence très similaire, à savoir l'implantation d'une fonction numérique *en précision fixée*. Un examen trop rapide pourrait laisser penser que les problèmes sont globalement les mêmes et qu'il n'y a pas grand chose à gagner du fait que la précision cible soit connue à l'avance. À part, à la marge, le fait de pouvoir déterminer *a priori* l'ordre de troncature et la précision de travail, on ne voit pas forcément ce qu'on pourrait gagner par rapport à l'utilisation du code générique multiprécision, instancié avec la précision cible désirée.

En réalité, et comme nous allons le voir dans ce chapitre, le fait de connaître la précision à l'avance permet, tant que celle-ci reste modérée, d'utiliser des approches complètement différentes et d'obtenir finalement un code bien plus efficace.

Admettons provisoirement qu'on puisse effectivement écrire un code bien plus efficace lorsqu'on connaît à l'avance la précision cible. Pourquoi, dans ce cas, avoir fait tant d'efforts pour réaliser une implantation efficace en précision arbitraire ? Les physiciens, les numériciens, n'ont besoin de connaître, en règle générale, que quelques dizaines de bits du résultat d'un calcul, tout au plus. Ne comblerait-on pas les besoins de tous les utilisateurs en fournissant une implémentation de chaque fonction numérique avec, disons, 200 bits corrects ? Auquel cas, une implémentation en précision arbitraire serait parfaitement inutile...

La réponse est non : nous n'avons pas perdu notre temps à produire une implémentation en précision arbitraire ; car pour écrire du code efficace en précision fixée, il est nécessaire de faire un grand nombre de calculs préliminaires. Ces calculs impliquent la fonction numérique que l'on souhaite implémenter et ils doivent évidemment être réalisés avec une précision supérieure à la précision cible. Et donc, bien que ça puisse paraître contre-intuitif, *pour pouvoir calculer efficacement une fonction numérique en précision donnée, il faut savoir au préalable calculer cette même fonction en précision plus grande.*

Un exemple tout à fait représentatif de la problématique d'implémentation en précision fixée est donné par les libms. Une libm (*mathematical library*) est une bibliothèque logicielle qui regroupe des implémentations des fonctions usuelles (exp, ln et leurs variantes dans différentes bases, fonctions trigonométriques et trigonométriques inverses, fonctions hyperboliques et hyperboliques inverses, quelques fonctions spéciales comme erf, les fonctions de Bessel, etc.). Elles sont généralement accessibles en C, par exemple, via l'inclusion du fichier `math.h`. Dans les libms usuelles, il n'y a pas de garantie sur la qualité d'approximation des fonctions. En règle générale, les implémentations fournissent une valeur approchée $y$ correspondant, peu ou prou, à une approximation de la valeur exacte $f(x)$ avec 53 bits de précision, ce qui correspond à la précision `double` de la norme IEEE-754. Cependant, cette règle générale n'a rien de systématique. Par exemple si on utilise la glibc (version 2.9) [1] pour calculer une valeur approchée de $\cos(2^{300})$, on obtient la valeur $0.962962\cdots$ qui n'a pas grand chose à voir avec la valeur exacte $\cos(2^{300}) = 0.21203\cdots$ Il s'agit pourtant de la libm distribuée en standard avec la plupart des distributions Linux.

Un des grands axes de travail de l'équipe Arénaire durant les dix dernières années a été de montrer qu'il était possible d'écrire une libm fournissant l'arrondi correct à la précision `double`, et dont le temps d'exécution moyen était du même ordre que celui des libms usuelles. Ce travail s'est concrétisé par l'écriture d'une bibliothèque mathématique appelée `CRlibm` [2]. Ce projet est né sous l'impulsion de Jean-Michel Muller et a été principalement coordonné par Florent de Dinechin. Le développement de `CRlibm` a alimenté plusieurs thèses à commencer par celle de David Defour [Def03] qui proposait alors l'arrondi correct *d'une* fonction (la fonction exp). La thèse de Guillaume Melquiond [Mel06] a été une importante pierre à l'édifice : bien que Melquiond n'ait pas travaillé directement au développement de `CRlibm`, il a écrit un logiciel appelé Gappa [3] qui a considérablement facilité l'analyse des erreurs d'arrondi dans les codes développés pour `CRlibm`. En outre, il a rendu ces analyses plus fiables en automatisant les calculs et en produisant des preuves formelles démontrant la correction des bornes calculées. La thèse de Christoph Lauter [Lau08] a marqué une avancée supplémentaire en automatisant une grande part du développement du code de `CRlibm`. Bien que je n'aie pas contribué directement au développement de `CRlibm`, les travaux présentés dans cette thèse se sont en grande partie nourris de problématiques liées à ce projet. Le développement du logiciel Sollya a été motivé initialement par un souci de simplification et d'automatisation des tâches nécessaires à la réalisation de `CRlibm`. De nombreux exemples de ce manuscrit sont donc naturellement tirés de ce travail.

1. Avec Linux 2.6.26-2 tournant sous Debian avec un Pentium D et gcc 4.3.3.
2. distribuée sous licence LGPL et disponible à l'adresse `http://lipforge.ens-lyon.fr/www/crlibm/`
3. distribué sous licences CeCILL et LGPL et disponible à l'adresse `http://lipforge.ens-lyon.fr/www/gappa/`

### 2.1.1 Développement d'une fonction en précision fixée

Nous allons à présent voir en quoi le développement d'une fonction pour une libm diffère de celui d'une fonction en précision arbitraire. Lorsqu'on ignore *a priori* la précision avec laquelle il faut calculer la fonction, on en est réduit à utiliser un procédé générique convergeant vers la valeur $f(x)$ à calculer : développement de Taylor ou développement asymptotique (c'est ce que nous avons vu sur l'exemple de erf et erfc) ou encore itération de Newton, développement en fraction continue, utilisation de la moyenne arithmético-géométrique, par exemple (voir par exemple [Bre80]).

À l'inverse, lorsqu'on connaît par avance la précision à laquelle on désire évaluer $f(x)$, il est possible de précalculer une fonction d'approximation fournissant la qualité désirée. Qui plus est, on peut se permettre d'explorer l'espace des possibles pour déterminer quel type d'approximation est le plus adapté, comment l'évaluer efficacement, avec quelle précision intermédiaire, etc.

Le développement d'une fonction pour une libm suit donc en règle générale le déroulement suivant :

1. Déterminer avec quelle précision $\varepsilon$ on souhaite approcher $f(x)$ ($\varepsilon$ mesure ici l'erreur relative).
2. Utiliser une technique de réduction d'argument pour se ramener à l'évaluation de $f$ (ou d'une fonction auxiliaire) sur un petit intervalle fermé $[a, b]$. Si une telle réduction d'argument n'est pas possible, découper le domaine de définition en autant de sous-intervalles $[a, b]$ que nécessaire.
3. Trouver une approximation $p$ de bonne qualité pour approcher $f$ sur l'intervalle $[a, b]$.
4. Écrire un code permettant d'évaluer $p$ efficacement.

Quelques commentaires s'imposent. La valeur de $\varepsilon$ retenue à l'étape 1 dépend du but que l'on se fixe. À tout le moins, on veut généralement fournir une valeur approchée finale dont la précision soit de l'ordre de la précision du format flottant utilisé pour représenter le résultat. Dans le cas d'une libm par exemple, il s'agit de la précision `double`. On prendra donc $\varepsilon$ légèrement plus petit que $2^{-53}$. Si l'on cherche à fournir très souvent l'arrondi correct, sans toutefois le garantir, il faut choisir $\varepsilon$ plus petit, par exemple de l'ordre de $2^{-65}$ (évidemment il faudra utiliser une arithmétique plus précise que la double précision pour faire les calculs ; nous y reviendrons). Dans ce cas, l'erreur commise entre la valeur calculée $y$ et la valeur exacte $f(x)$ se situe au niveau du $65^{\text{e}}$ bit et il est donc probable que les 53 premiers bits ne seront pas affectés par cette légère erreur. Pour autant, on ne peut en être certain : si par exemple la valeur exacte est

$$f(x) = 0, b_1 \cdots b_{53} 1000000000001,$$

la valeur calculée pourra être par exemple

$$y = 0, b_1 \cdots b_{53} 0111111111111.$$

Dans ce cas, l'erreur entre la valeur calculée $y$ et la valeur exacte $f(x)$ est bien inférieure à $2^{-65}$ mais arrondir $y$ au `double` le plus proche conduit à $0, b_1 \cdots b_{53}$ alors que l'arrondi exact de $f(x)$ en arrondi au plus près est le nombre flottant immédiatement supérieur. Il s'agit du dilemme du fabricant de tables que nous avons déjà rencontré au chapitre 1 (voir figure 1.3 page 39). Ce phénomène se produit lorsque la valeur exacte est très proche du milieu de deux nombres flottants (dans le cas de l'arrondi au plus près). Dans ce cas, la propagation d'une retenue depuis les bits de poids faible vers les bits de poids forts vient influer sur les 53 premiers bits finalement retenus. Pour être dans la situation de dilemme il faut que le développement binaire de $f(x)$ ait une forme bien particulière, c'est-à-dire qu'à partir du $54^{\text{e}}$ bit il soit de la forme $100\cdots001$ ou $011\cdots110$. Il s'agit donc d'un phénomène plutôt rare (on s'attend intuitivement à ce que sa probabilité soit divisée par deux, chaque fois qu'on augmente la précision d'un bit, en faisant l'hypothèse — pas toujours vraie cependant — que les bits de $f(x)$ suivent une distribution aléatoire uniforme). En

prenant $\varepsilon = 2^{-65}$, on s'attend donc à ce que le cas défavorable se produise environ une fois sur $2^{11}$. Autrement dit, l'arrondi est correct dans plus de $99,9\%$ des cas.

Si on veut, comme `CRlibm`, garantir l'arrondi correct dans *tous* les cas, il faut être plus soigneux. Le cas où $f(x)$ est exactement le milieu de deux nombres flottants est évidemment possible. Pour certaines fonctions comme exp ou cos, le théorème de Lindemann-Weierstrass [Bak75, Chap. 1] permet de montrer que de tels cas exacts n'existent pas (à part pour $x = 0$). Pour les autres fonctions (comme erf par exemple) on ne possède pas de résultat théorique. Cependant, puisqu'il n'y a qu'un nombre fini de valeurs de $x$, une approche exhaustive est envisageable en théorie. Mais une telle stratégie est déraisonnable en pratique, compte tenu du grand nombre de valeurs possibles pour $x$.

Vincent Lefèvre a développé dans sa thèse [Lef00] un algorithme qui permet d'accélérer cette approche exhaustive. Son algorithme élimine au fur et à mesure de grandes plages de valeurs de $x$ pour lesquelles il est possible de montrer que $f(x)$ est relativement éloigné du milieu de deux nombres flottants. Dans le cas du format `double`, il peut, en quelques mois de calculs distribués sur plusieurs machines, trouver tous les cas exacts. Dans l'implantation de la fonction $f$, ces cas exacts sont traités à part.

L'algorithme de Lefèvre fait mieux que de trouver les cas exacts : il trouve les *pires cas*, c'est-à-dire les cas pour lesquels la distance entre $f(x)$ et le milieu de deux nombres flottants est minimale (parmi les cas non exacts). Notons $\varepsilon$ cette distance minimale. On suppose que $f(x)$ n'est pas un cas exact. Alors, si $y$ est une valeur approchée de $f(x)$ à moins de $\varepsilon$, on est sûr que le phénomène de propagation de retenue néfaste ne peut pas se produire. Dans ce cas $y$ et $f(x)$ s'arrondissent à la même valeur : l'arrondi de $y$ est l'arrondi correct de $f(x)$. Pour le format `double`, on trouve généralement une valeur de $\varepsilon$ de l'ordre de $2^{-120}$. La détermination des pires cas demeure un problème difficile qui fait encore l'objet de recherches actives [LSZ08] pour pouvoir traiter des précisions supérieures au format `double`.

Les points 2 et 3 du schéma général présenté plus haut ne sont pas forcément indépendants : en général, plus l'intervalle est petit, plus la fonction d'approximation est simple et peut donc être évaluée rapidement. Mais plus l'intervalle est petit, plus la réduction elle-même est coûteuse (en temps de calcul ou en mémoire nécessaire pour stocker des données tabulées). Il y a donc un compromis à trouver entre les deux.

Concernant le point 3, dans la très grande majorité des cas, la fonction $p$ sera un polynôme. La majeure partie de ce chapitre sera consacrée à décrire des outils permettant d'obtenir des polynômes d'approximation vérifiant toutes sortes de contraintes souhaitables pour approcher $f$. Mais il est parfois plus intéressant de choisir d'autres types d'approximation : polynômes trigonométriques ou fraction rationnelle par exemple. Le cas des fractions rationnelles sera abordé en fin de chapitre, en section 2.4.

L'évaluation de l'approximation $p$ effectuée au point 4 constitue un domaine de recherche en soi. On choisit souvent une approximation polynomiale : en effet, les polynômes peuvent être évalués en n'utilisant que des additions et des multiplications, qui sont les opérations les plus rapides disponibles sur les processeurs actuels. Pour ce qui est des polynômes, plusieurs schémas d'évaluation ont été étudiés en détail (le livre de Knuth [Knu98, Chap. 4.6.4] offre une bonne introduction à ce sujet ; on peut aussi consulter [Rev06]). Ils permettent d'utiliser au mieux le parallélisme disponible sur l'architecture visée. Quel que soit le schéma d'évaluation utilisé, il faut se préoccuper des erreurs d'arrondi qui se propagent au cours de l'algorithme. Cette analyse d'erreur étant assez pénible à effectuer, elle gagne à être automatisée lorsque c'est possible. Une telle automatisation implique généralement de se limiter à un ou deux schémas d'évaluation bien balisés. Par exemple, dans sa thèse [Lau08], Christoph Lauter se limite à la génération automatique de code évaluant $p$ par le schéma de Horner.

On peut imaginer qu'il y ait une interaction entre le point 3 et le point 4 : certains schémas

d'évaluation imposent de récrire le polynôme $p$ sous une certaine forme. Si on sait qu'on va utiliser tel schéma d'évaluation particulier, il peut donc être intéressant de spécifier quelques contraintes sur la structure du polynôme $p$ qu'on recherche. Ceci induit donc possiblement un va-et-vient entre la recherche du polynôme et son implantation pratique.

Les efforts conjugués de la thèse de Lauter et de la présente thèse ont permis d'automatiser une très grande partie des points 3 et 4. Cette automatisation rend désormais possible une exploration systématique de tous les compromis *réduction d'argument/qualité de l'approximation* par essais successifs : on choisit une réduction d'argument, on en déduit automatiquement (et donc en quelques dizaines de secondes) une implémentation, on en mesure les performances, et on recommence avec une autre réduction d'argument. Ceci permet de choisir le compromis qui est effectivement le meilleur en pratique. De plus, compte tenu de la rapidité du processus, on peut même imaginer choisir un compromis qui sera optimal *sur une architecture particulière*, et qui n'est pas forcément le même que celui qu'on aurait sur une autre architecture (avec des tailles de cache, fréquences de bus, latences d'opérations élémentaires, caractéristiques de parallélisme, etc. différentes).

### 2.1.2 Expansions de nombres flottants

Lors de l'évaluation de $p$ à la phase 4, il faut nécessairement utiliser une arithmétique fournissant plus de précision que l'arithmétique double précision. En effet, si l'on souhaite obtenir finalement une valeur approchée à $\varepsilon$ près ($\varepsilon \leq 2^{-65}$ voire $\varepsilon \leq 2^{-120}$) on ne peut se contenter d'utiliser directement l'arithmétique flottante proposée par le matériel : la moindre opération créera une erreur d'arrondi de l'ordre de $2^{-53}$ qui surpassera largement l'erreur totale admissible. Il faut donc travailler, d'une façon ou d'une autre, avec une précision supérieure à la précision machine.

On pourrait utiliser une bibliothèque multiprécision telle que `MPFR` par exemple, mais on perdrait alors quasiment tout l'avantage de connaître la précision cible à l'avance : en effet, la généricité d'une telle bibliothèque la rend particulièrement peu adaptée à des précisions faibles. On paie un très gros surcoût, dû à la gestion de la multiprécision, par rapport à l'arithmétique directement implémentée dans le processeur.

Une possibilité consiste à récrire une bibliothèque du type de `MPFR`, mais plus optimisée que `MPFR` pour des précisions d'une centaine de bits. C'est la solution qui était adoptée dans `CRlibm` au début de son développement. L'évaluation de $p$ était alors effectuée avec SCSLib [DdD02] (pour *Software Carry-Save library*). Mais là encore, une telle bibliothèque utilisant l'arithmétique entière du processeur, il faut payer un surcoût non négligeable du fait des conversions *arithmétique flottante/arithmétique entière* qu'il faut constamment faire. À l'heure actuelle, cette bibliothèque n'est plus du tout utilisée dans `CRlibm`.

La solution actuellement retenue est celle des expansions de flottants : on représente les nombres par un couple de nombres flottants $(x_h, x_\ell)$. Un tel couple (appelé double-`double`) représente, en fait, le nombre $a = x_h + x_l$. Un nombre représenté sous cette forme est appelé un double-`double`. Il est aisé de s'apercevoir que tout nombre flottant de 107 bits est représentable de cette manière. Des algorithmes existent pour additionner et multiplier des doubles-`double` [Dek71]. On peut ainsi simuler grossièrement un doublement de la précision machine.

Pour atteindre la précision de 120 bits, qui correspond généralement au pire cas du dilemme du fabricant de tables, un simple doublement de la précision ne suffit pas. Pour cela, on peut utiliser la même stratégie, mais en utilisant des triplets au lieu de couples : $(x_h, x_m, x_l)$ représente alors le nombre $a = x_h + x_m + x_l$. On parle alors de triple-`double`. La manipulation efficace des triples-`double` est un peu plus subtile et coûteuse que celle des doubles-`double`, mais là encore, des algorithmes permettent d'additionner/multiplier des triples-`double` [Lau05].

Comme on l'a vu, un problème fondamental dans le développement d'une fonction pour une libm consiste à trouver une bonne approximation (souvent polynomiale). Elle doit être à la fois rapide à évaluer (pour un polynôme, cela revient généralement à minimiser son degré mais aussi

la taille en bits de ses coefficients) et suffisamment précise. Dans toute la suite du chapitre, nous allons étudier ce problème en détail.

## 2.2 Meilleure approximation polynomiale

Nous nous intéressons tout d'abord à la recherche d'approximations polynomiales. À moins que le polynôme ait une forme très particulière qui permette de concevoir un algorithme d'évaluation *ad hoc*, le temps nécessaire à l'évaluation d'un polynôme est directement lié à son degré : plus celui-ci est élevé, plus l'évaluation du polynôme nécessite d'opérations et donc plus le temps d'évaluation est grand. Dans toute la suite, nous supposerons donné le degré $n$ du polynôme recherché.

Parfois, il peut être utile de fournir des spécifications plus précises : par exemple, pour approcher une fonction paire, il peut être raisonnable de chercher un polynôme pair plutôt qu'un polynôme quelconque. Ce faisant, on augmentera peut-être le degré du polynôme mais en annulant la moitié des coefficients ; de ce fait, le polynôme pair peut être en réalité bien plus intéressant que le polynôme quelconque. De même, on peut souhaiter spécifier la valeur de certains coefficients. Par exemple, le développement en série de $\exp(x)$ en 0 est $\exp(x) = 1 + x + x^2/2 + \cdots$ Plutôt que de chercher un polynôme de degré $n$ quelconque pour approcher exp, il peut être raisonnable de le chercher sous la forme

$$p(x) = 1 + x + \frac{x^2}{2} + a_3 x^3 + \cdots + a_n x^n.$$

Pour évaluer ce polynôme, on économise des multiplications et donc on gagne du temps par rapport à l'évaluation d'un polynôme quelconque.

Comme nous le verrons, le problème de l'approximation avec contraintes sur les coefficients se ramène à un calcul de polynôme de meilleure approximation exprimé dans une base « à trous », c'est-à-dire une base qui ne contient pas tous les monômes (la base $x^3, \ldots, x^n$ dans l'exemple précédent). L'approximation à trous est reliée à une propriété appelée *condition de Haar*. Nous nous pencherons d'abord sur le problème de l'approximation polynomiale classique avant d'étudier la condition de Haar et l'approximation à trous en section 2.2.3.

### 2.2.1 Cas classique

L'idée d'approcher une fonction continue par un polynôme sur un intervalle compact $[a, b]$ trouve sa légitimité dans le théorème de Weierstrass qui affirme qu'il est possible d'approcher uniformément une fonction continue d'aussi près qu'on veut par un polynôme, pourvu que son degré soit suffisamment élevé.

**Théorème 2.1** (Weierstrass)**.** *On note $\mathcal{C}([a, b])$ l'ensemble des fonctions continues de $[a, b]$ dans $\mathbb{R}$ muni de la norme sup :*

$$\|g\|_\infty = \max_{x \in [a,b]} |g(x)|.$$

*Alors, l'ensemble des polynômes à coefficients réels est dense dans $\mathcal{C}([a, b])$. En d'autres termes, pour toute fonction $f \in \mathcal{C}([a, b])$, il existe une suite $(p_n)$ de polynômes telle que $\|p_n - f\|_\infty \to 0$.*

*Démonstration.* Le livre de Cheney [Che82, Chap. 3] fournit une preuve de ce théorème ainsi que des références bibliographiques très complètes sur son histoire. □

Il faut noter que, si l'intervalle $[a, b]$ n'était pas borné, le résultat serait faux. Par exemple, pour tout polynôme $p$, quel que soit son degré, on a $\|\sin - p\|_\infty^{[0,+\infty[} \geq 1$. La phase de réduction d'argument est donc indispensable si on souhaite travailler avec une approximation polynomiale. Si $f$ ne présente pas d'identité remarquable permettant une réduction d'argument, et qu'elle est néanmoins définie sur un intervalle non borné, il faut généralement recourir à d'autres formes

d'approximation (fraction rationnelle par exemple), voire revenir à des stratégies du même type que celles mises en place pour la précision arbitraire (utilisation d'un développement en série de Taylor par exemple).

En règle générale, nous nous intéressons plus à l'erreur relative qu'à l'erreur absolue (même si pour certaines applications, l'erreur absolue sera privilégiée). Dans ce cas, le théorème de Weierstrass n'est plus vrai : il existe des fonctions continues sur $[a, b]$ qui ne peuvent être arbitrairement approchées par un polynôme, en erreur relative. Cependant, les fonctions usuelles ne sont pas si pathologiques.

Si $f$ ne s'annule pas sur $[a, b]$, par exemple, elle peut être approchée d'aussi près que l'on veut en erreur relative. En effet, considérons une suite de polynômes $(p_n)$ telle que $\|p_n - f\|_\infty$ tende vers 0. Une telle suite existe d'après le théorème de Weierstrass. Il est facile de montrer que $\|p_n/f - 1\|_\infty$ converge aussi vers 0, donc on peut approcher $f$ d'aussi près qu'on le désire en erreur relative.

**Remarque 2.2.** *Soient $p = \sum a_j x^j$ un polynôme de degré $n$ et $f$ une fonction continue qui ne s'annule pas sur $[a, b]$. L'erreur relative entre $p$ et $f$ est*

$$\frac{p-f}{f} = \frac{p}{f} - 1 = \left(\sum_{j=0}^{n} a_j \frac{x^j}{f(x)}\right) - 1.$$

*Si on définit $\varphi_j : x \mapsto x^j/f(x)$ et $g : x \mapsto 1$, l'erreur relative entre $p$ et $f$ est donc égale à*

$$\left(\sum_{j=0}^{n} a_j \varphi_j\right) - 1.$$

*Il y a donc équivalence entre la recherche d'un polynôme approchant $f$ en erreur relative et la recherche d'une combinaison linéaire des $\varphi_j$ approchant $g$ en erreur absolue.*

Il est fréquent que $f$ s'annule dans l'intervalle. Mais en règle générale, elle n'a qu'un nombre fini de zéros, tous d'ordre fini, dans l'intervalle (en fait, la plupart du temps $f$ est même analytique, ce qui garantit cette propriété). Pour simplifier, supposons que $f$ a un unique zéro d'ordre fini $k$ en un point $x_0$ de l'intervalle. Dans ce cas, pour que l'erreur relative entre le polynôme et la fonction reste bornée au voisinage de $x_0$, il est nécessaire que $p$ ait aussi un zéro d'ordre au moins $k$ en $x_0$. Donc le polynôme doit être de la forme $a_k (x-x_0)^k + \cdots + a_n (x-x_0)^n$. Chercher à approcher $f(x)$ en erreur relative par un polynôme de cette forme est équivalent à chercher un polynôme de la forme $a_k + a_{k+1}(x-x_0) + \cdots + a_n (x-x_0)^{n-k}$ approchant en erreur relative $f(x)/(x-x_0)^k$ (qui ne s'annule pas sur l'intervalle). Nous sommes donc dans le cas précédent et nous savons que, quitte à augmenter $n$, nous pouvons approcher $f$ en erreur relative d'aussi près que nous voulons.

**Remarque 2.3.** *Si on définit $\varphi_j : x \mapsto (x-x_0)^{k+j}/f(x)$ et $g : x \mapsto 1$, le problème d'approximation de $f$ en erreur relative par un polynôme de degré $n$ devient le problème d'approximation de $g$ par une combinaison linéaire de $\varphi_0, \ldots, \varphi_{n-k}$ en erreur absolue.*

La première question qu'il est naturel de se poser est de savoir, $n$ étant donné, quelle est la plus petite erreur que nous puissions obtenir. Compte tenu des remarques précédentes, nous allons étudier cette question dans un cadre un peu général. On suppose données $n+1$ fonctions continues $\varphi_0, \ldots, \varphi_n$ et une autre fonction continue $g$ ; on cherche à savoir quelle est l'erreur absolue minimale entre une combinaison linéaire des $\varphi_j$ (un « polynôme ») et $g$. On pourra garder en tête les cas

typiques $\varphi_j(x) = x^j$ et $g = f$ (cas de l'approximation en erreur absolue) ou $\varphi_j(x) = (x-x_0)^{k+j}/f(x)$ et $g = 1$ (cas de l'erreur relative) ; mais pour tout le reste du chapitre, nous raisonnerons en toute généralité avec des fonctions $\varphi_j$ et $g$ continues quelconques. Une combinaison linéaire des $\varphi_j$ pourrait être appelée un *polynôme généralisé*. Cependant, puisque dans toute la suite nous nous intéressons exclusivement à ces polynômes généralisés, nous avons jugé plus simple de parler de « polynômes » en mettant le terme entre guillemets pour rappeler qu'il ne s'agit pas de l'acception usuelle du mot.

**Proposition 2.4** (Existence de meilleures approximations)**.** *Parmi tous les « polynômes » (c'est-à-dire les combinaisons linéaires de $\varphi_0, \ldots, \varphi_n$), il existe au moins un « polynôme » $p$ tel que la norme $\|p-g\|_\infty$ soit minimale. Un tel « polynôme » est dit* de meilleure approximation *ou encore* minimax *(il minimise le maximum de la valeur absolue de l'erreur).*

*Démonstration.* Il s'agit d'un résultat classique. L'ensemble $S = \{\|g - \sum a_j\varphi_j\|_\infty, (a_0, \ldots, a_n) \in \mathbb{R}^{n+1}\}$ est un sous-ensemble des nombres réels positifs. Il admet donc une borne inf $\mu$. Soit alors $(p_k)$ une suite de « polynômes » telle que $\|p_k - g\|_\infty$ tende vers $\mu$ en décroissant lorsque $k \to +\infty$. On a pour tout $k$,

$$\|p_k\|_\infty \leq \|p_k - g\|_\infty + \|g\|_\infty \leq \|p_0 - g\|_\infty + \|g\|_\infty .$$

La suite $(p_k)$ est donc bornée dans un espace de dimension finie. Elle admet donc une valeur d'adhérence $p^\star$. Par continuité de la norme, $\|p^\star - g\|_\infty = \mu$. □

Pour un degré $n$ donné, il existe donc une erreur $\mu_n$ minimale. Si l'erreur cible $\varepsilon$ qu'on désire atteindre est inférieure à $\mu_n$, il faut impérativement augmenter le degré pour espérer approcher la fonction $f$ à moins de $\varepsilon$. Nous reviendrons un peu plus tard sur la détermination du plus petit degré $n$ tel que $\mu_n \leq \varepsilon$. Pour l'instant, nous supposons $n$ donné et nous allons désormais chercher à caractériser et calculer les « polynômes » de meilleure approximation.

Lorsque $\varphi_j(x) = x^j$ (approximation polynomiale au sens usuel), il y a un unique polynôme de meilleure approximation $p^\star$ qui est caractérisé par le théorème d'alternance [4] :

**Théorème 2.5** (d'alternance)**.** *Soit $g$ une fonction continue sur $[a, b]$. Un polynôme $p = \sum_{j=0}^{n} a_j x^j$ est de meilleure approximation pour $g$ (parmi les polynômes de degré inférieur ou égal à $n$) si et seulement s'il existe $n+2$ points $x_0 < x_1 < \cdots < x_{n+1}$ dans $[a, b]$ et $\mu \in \mathbb{R}$ tels que :*

$$\begin{cases} \forall i \in [\![0, n+1]\!], \quad p(x_i) - g(x_i) = (-1)^i \mu \\ |\mu| = \|p - g\|_\infty \ . \end{cases} \tag{2.1}$$

*Démonstration.* La situation décrite par le système (2.1) est illustrée par la figure 2.1. Le livre de Cheney [Che82] en donne une preuve élégante mais peu intuitive. Une preuve plus élémentaire et intuitive se trouve dans [Ste98b]. □

**Corollaire 2.6** (Unicité du polynôme de meilleure approximation)**.** *Sous les mêmes hypothèses que le théorème 2.5, le polynôme de meilleure approximation est unique.*

*Démonstration.* Voir [Ste98b] pour une preuve de ce théorème. □

Le théorème d'alternance affirme qu'un polynôme est optimal si et seulement si son erreur oscille parfaitement entre ses valeurs extrêmes au moins $n+2$ fois. Le théorème de La Vallée Poussin, que nous allons voir à présent, raffine le théorème 2.5. Il affirme la chose suivante : soit un polynôme $p$ dont l'erreur oscille au moins $n+2$ fois, mais imparfaitement. Alors, la qualité d'approximation de $p$ est reliée à la qualité des oscillations de sa fonction d'erreur.

4. Ce théorème est parfois appelé *théorème de Tchebychev*. D'après Cheney [Che82], il est énoncé pour la première fois par Borel [Bor05]. Cependant, les travaux de Tchebychev sur ce sujet sont précurseurs : beaucoup d'idées sont déjà présentes dans [Tch59] ; il démontre en particulier que, si $p$ est optimal, il existe $n+2$ points $x_0 < x_1 < \cdots < x_{n+1}$ tels que $\forall i \in [\![0, n+1]\!]$, $|p(x_i) - g(x_i)| = \|p - g\|_\infty$.

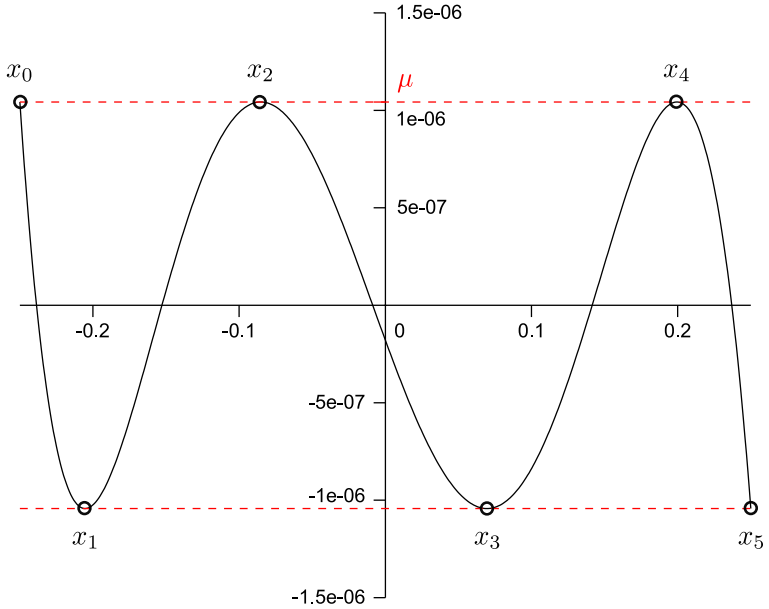



**Figure 2.1 :** Graphe de l'erreur entre $\sin(x)/\ln(1+x)$ et son polynôme de meilleure approximation de degré 4 sur $[-1/4, 1/4]$. L'erreur maximale est atteinte 6 fois avec alternance de signe.

**Théorème 2.7** (La Vallée Poussin)**.** *On se place sous les mêmes hypothèses que pour les théorèmes précédents. On suppose donné un polynôme $p$ et on suppose qu'il existe $n+2$ points $x_0 < x_1 < \cdots < x_{n+1}$ dans $[a, b]$ et $\varepsilon \in \mathbb{R}$ tels que :*

$$\forall i \in [\![0,\, n+1]\!]\,, \quad p(x_i) - g(x_i) = (-1)^i \varepsilon.$$

*On note $\mu$ l'erreur optimale. Alors*

$$|\varepsilon| \leq \mu \leq \|p - g\|_\infty\,.$$

*Démonstration.* La situation décrite dans ce théorème est illustrée par la figure 2.2. On pourra se reporter à [Ste98b] pour une preuve du théorème. □

Le polynôme $p$ étant donné, on a tout intérêt à choisir les points $x_i$ de telle sorte que $\varepsilon$ soit maximal. Ceci fournit une formulation équivalente du théorème de La Vallée Poussin.

**Théorème 2.8** (La Vallée Poussin, autre formulation)**.** *Toujours sous les mêmes hypothèses, on suppose qu'il existe $n+2$ points $x_0 < x_1 < \cdots < x_{n+1}$ où l'erreur entre $p$ et $g$ a un extremum local et tels que le signe de l'erreur alterne entre deux $x_i$ successifs. On suppose en outre que l'extremum global est atteint en l'un des $x_i$. Alors l'erreur optimale $\mu$ vérifie*

$$\min_{i \in [\![0,\, n+1]\!]} |p(x_i) - g(x_i)| \;\leq\; \mu \;\leq\; \max_{i \in [\![0,\, n+1]\!]} |p(x_i) - g(x_i)|.$$

*Démonstration.* La preuve est facile et laissée au lecteur. Il s'agit d'un usage répété du théorème des valeurs intermédiaires. □

Outre le fait qu'il énonce un résultat profond (l'amplitude des oscillations est reliée à la qualité d'approximation), ce théorème fournit un moyen effectif de mesurer la qualité d'approximation d'un polynôme. En effet, $p$ étant donné, on peut calculer $\varepsilon$ et $\|p - g\|_\infty$. Si ces deux valeurs sont suffisamment proches l'une de l'autre, $p$ peut être considéré comme pratiquement optimal.

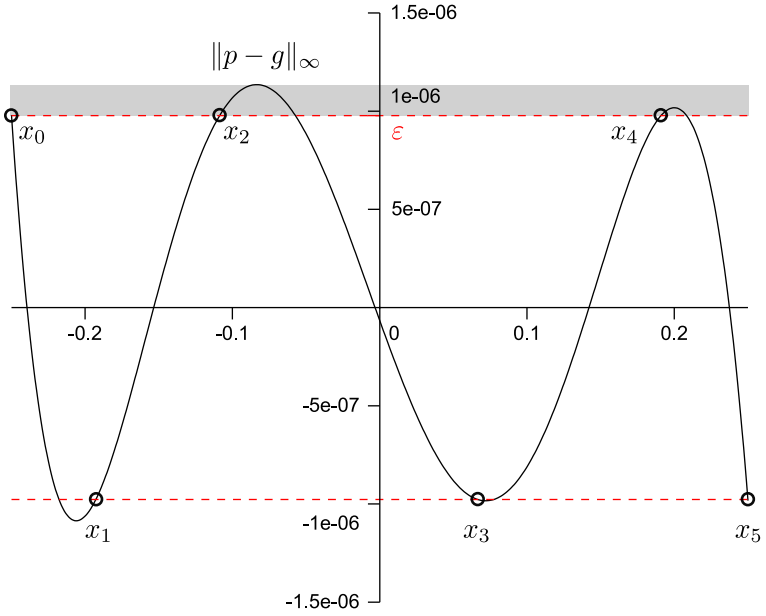



**Figure 2.2 :** Graphe de l'erreur entre $\sin(x)/\ln(1+x)$ et un polynôme non optimal de degré 4 sur $[-1/4, 1/4]$. L'erreur optimale se situe dans la bande grise.

En d'autres termes, nous pouvons voir $\|p-g\|_\infty$ comme une approximation de l'erreur optimale $\mu$. L'erreur relative entre les deux valeurs peut être majorée en fonction de $\varepsilon$ :

$$\left|\frac{\|p-g\|_\infty - \mu}{\mu}\right| \leq \frac{\|p-g\|_\infty - |\varepsilon|}{|\varepsilon|} = \delta. \tag{2.2}$$

La valeur $\delta$ ainsi définie mesure la qualité d'approximation de $p$. Si par exemple $\delta \leq 10^{-5}$, cela signifie que $\|p-g\|_\infty$ (ou $|\varepsilon|$) est une estimation de l'erreur optimale avec approximativement cinq décimales correctes.

Nous avons à présent les outils pour comprendre ce que faisait le petit script d'exemple de Sollya présenté en page 73. Dans ce script, on cherche à approcher la fonction exp par un polynôme de degré 5 sur $[0, 1]$. On commence par invoquer la commande `remez` (que nous décrirons un peu plus tard) pour calculer un polynôme $p$ dont l'erreur est quasi-optimale. Plus précisément — en reprenant les notation de l'équation (2.2) — on demande que l'erreur relative entre $\|p-\exp\|_\infty$ et $\mu$ soit inférieure au paramètre `qualiteDemandee` (il suffit pour cela que $\delta \leq$ `qualiteDemandee`). Une fois $p$ calculé, le script fait l'hypothèse que l'erreur $p-\exp$ a exactement $n+2$ extrema locaux et qu'ils sont alternés. Cette hypothèse est vérifiée dans cet exemple (elle n'a aucune raison de l'être en général ; il faudrait donc adapter le script pour qu'il fonctionne en toute généralité, mais cela dépasse le cadre de cet exemple). Les points $x_i$ correspondant aux extrema locaux sont déterminés de façon approchée par un algorithme numérique (`dirtyfindzeros`) et stockés dans la liste `ptsCritiques`. Le minimum et le maximum des $|p(x_i) - \exp(x_i)|$ sont alors calculés. Le théorème de La Vallée Poussin permet d'affirmer que $\mu \in$ [`minimum`, `maximum`]. En affichant cet intervalle avec le mode `midpointmode` de Sollya, on voit immédiatement les premiers chiffres significatifs de $\mu$. En outre, on peut calculer le paramètre $\delta$ (stocké dans la variable `qualiteEffective`) et observer que l'erreur relative entre $\|p-\exp\|_\infty$ et $\mu$ est bornée par $1.7\,\mathrm{e}{-16}$ et est donc bien inférieure à `qualiteDemandee`.

Revenons à présent au cas général qui nous intéresse : on recherche une combinaison linéaire de fonctions de base $\varphi_j$ la plus proche possible de $g$. En réalité, les preuves des théorèmes 2.5, 2.6 et 2.7 ne s'appuient que sur quelques propriétés vérifiées par les polynômes de degré inférieur ou égal à $n$ :

— étant donnés $n+1$ points distincts $x_0 < \cdots < x_n$ et $n+1$ valeurs (distinctes ou non) $y_0, \ldots, y_n$, il existe un unique polynôme interpolateur des valeurs $y_i$ aux points $x_i$ ;
— deux polynômes qui coïncident en $n+1$ points sont égaux ;
— un polynôme a au plus $n$ racines ;
— étant donnés $n$ points distincts $z_1 < \cdots < z_n$, il existe un unique polynôme (à un facteur multiplicatif près) qui a les $z_i$ pour racines.

Ceci conduit à énoncer une condition, appelée *condition de Haar*, qui généralise ces propriétés au cas des combinaisons linéaires de fonctions de base $\varphi_0, \ldots, \varphi_n$.

**Définition 2.9** (Condition de Haar)**.** On dit que les fonctions de base $\varphi_0, \ldots, \varphi_n$ vérifient la condition de Haar sur $[a, b]$ lorsque les conditions (équivalentes) suivantes sont vérifiées :

(i) Pour tout choix de $n+1$ points distincts $x_0 < \cdots < x_n$ dans $[a, b]$, le déterminant suivant est non nul :
$$\begin{vmatrix} \varphi_0(x_0) & \cdots & \varphi_n(x_0) \\ \vdots & \vdots & \vdots \\ \varphi_0(x_n) & \cdots & \varphi_n(x_n) \end{vmatrix} \neq 0.$$
(ii) Étant donnés $n+1$ points distincts $x_0 < \cdots < x_n$ et $n+1$ valeurs (distinctes ou non) $y_0, \ldots, y_n$, il existe un unique « polynôme » $\sum_{j=0}^{n} a_j \varphi_j$ qui interpole les valeurs $y_i$ aux points $x_i$.
(iii) Étant donnés $n$ points distincts $z_1 < \cdots < z_n$, il existe un unique « polynôme » (à un facteur multiplicatif près) qui a les $z_i$ pour racines.
(iv) Deux « polynômes » qui coïncident en $n+1$ points ont tous leurs coefficients égaux.
(v) Si $p = \sum_{j=0}^{n} a_j \varphi_j$ et que les $a_j$ ne sont pas tous nuls, $p$ a au plus $n$ zéros dans $[a, b]$.

*Démonstration.* On trouvera dans [BE95] la définition ainsi que de nombreuses propositions relatives à la condition de Haar. Par soucis de complétude, nous justifions ici l'équivalence des cinq points.

**(i) ⇒ (ii) :** un « polynôme » $p = \sum_{j=0}^{n} a_j \varphi_j$ interpole les $y_i$ aux points $x_i$ si et seulement si les coefficients $a_j$ sont solutions du système :
$$\begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_n(x_0) \\ \vdots & \vdots & \vdots \\ \varphi_0(x_n) & \cdots & \varphi_n(x_n) \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}.$$
Ce système est inversible par hypothèse et admet donc une unique solution.

**(ii) ⇒ (iii) :** soient $z_1, \ldots, z_n$ des points distincts de $[a, b]$ et soit $\alpha$ un $(n+1)$-ième point. Soit $p$ l'unique « polynôme » qui vaut 0 aux points $z_i$ et qui vaut 1 au point $\alpha$. Supposons que $q$ est un autre « polynôme » s'annulant aux points $z_i$. Alors $q(\alpha)p$ est un « polynôme » qui s'annule aux points $z_i$ et vaut $q(\alpha)$ en $\alpha$. Par unicité, on a donc $q(\alpha)p = q$ et $q$ est bien un multiple de $p$.

**(iii) ⇒ (iv) :** soient $p$ et $q$ deux « polynômes » coïncidant en $n+1$ points distincts $z$, $z_1, \ldots, z_n$. Alors $p - q$ s'annule aux points $z_i$. Donc $p - q$ est un multiple du « polynôme » $r$ valant 1 en $z$ et 0 aux points $z_i$. Comme $p - q$ s'annule en $z$, $p - q = 0r$, donc les coefficients de $p$ et $q$ sont égaux.

**(iv) ⇒ (v) :** supposons que $p = \sum_{j=0}^{n} a_j \varphi_j$ s'annule en $n+1$ points distincts $z_0 < \cdots < z_n$ de $[a, b]$. Par unicité, si $q = \sum_{j=0}^{n} b_j \varphi_j$ s'annule aussi aux points $z_i$, on a $\forall j$, $a_j = b_j$. En prenant $q$ identiquement nul, $b_j = 0$ et donc $a_j = 0$ pour tout $j$.

**(v) ⇒ (i) :** par contraposée. Supposons qu'il existe $n+1$ points distincts $x_0 < \cdots < x_n$ dans $[a, b]$ tels que

$$\begin{vmatrix} \varphi_0(x_0) & \cdots & \varphi_n(x_0) \\ \vdots & \vdots & \vdots \\ \varphi_0(x_n) & \cdots & \varphi_n(x_n) \end{vmatrix} = 0.$$

Alors il existe une relation non triviale entre les colonnes de la matrice :

$$\exists(\lambda_0, \ldots, \lambda_n) \neq (0, \ldots, 0) \quad \text{tel que} \quad \forall i \in [\![0, n]\!], \sum_{j=0}^{n} \lambda_j \varphi_j(x_i) = 0.$$

Il suit que $\sum_{j=0}^{n} \lambda_j \varphi_j$ est un « polynôme » avec au moins $n+1$ racines distinctes (les $x_i$) et dont les coefficients ne sont pas tous nuls. □

**Remarque 2.10.** *On suppose que la condition de Haar est remplie. Soient $z_1 < \cdots < z_n$ des points distincts de $[a, b]$. On définit la fonction $p$ par*

$$p : z \mapsto \begin{vmatrix} \varphi_0(z) & \cdots & \varphi_n(z) \\ \varphi_0(z_1) & \cdots & \varphi_n(z_1) \\ \vdots & \vdots & \vdots \\ \varphi_0(z_n) & \cdots & \varphi_n(z_n) \end{vmatrix}.$$

*En développant le déterminant par rapport à la première ligne, on voit que $p(z)$ s'exprime comme une combinaison linéaire des $\varphi_j(z)$ dont les coefficients sont des mineurs du système. Donc $p$ est un « polynôme ». Par construction, $p$ s'annule en $z_1, \ldots, z_n$ et ne s'annule en aucun autre point. Il s'agit donc de l'unique « polynôme » (à un facteur multiplicatif près) qui s'annule aux points $z_i$. On peut montrer que $p$ s'annule en changeant de signe en chaque $z_i$.*

On parle parfois de *système de Tchebychev* pour parler d'une famille $(\varphi_0, \ldots, \varphi_n)$ vérifiant la condition de Haar, et *d'ensemble de Tchebychev* pour désigner l'espace vectoriel engendré par les $\varphi_j$.

La base des monômes $1, x, \ldots, x^n$ est le prototype d'un système qui vérifie la condition de Haar. On en déduit immédiatement que d'autres systèmes usuels vérifient cette condition.

**Exemple 2.11.** *Soit $f$ une fonction qui ne s'annule pas dans $[a, b]$. Alors, le système $1/f(x), x/f(x), \ldots, x^n/f(x)$ vérifie la condition de Haar.*

De la même façon, si $f$ a un unique zéro d'ordre $k$, le système de fonctions que nous avons défini plus haut vérifie la condition de Haar.

**Exemple 2.12.** *Soit $f$ une fonction admettant un unique zéro $x_0$ d'ordre fini $k$ dans $[a, b]$. Alors, le système $(x - x_0)^k/f(x), (x - x_0)^{k+1}/f(x), \ldots, (x - x_0)^n/f(x)$ vérifie la condition de Haar.*

Toute la théorie classique se généralise au cas où les fonctions de base $\varphi_0, \ldots, \varphi_n$ vérifient la condition de Haar : théorème d'alternance, unicité, théorème de La Vallée Poussin.

**Exemple 2.13.** *Il est possible de faire de l'approximation avec autre chose que des polynômes. Par exemple, si $\lambda_0 < \lambda_1 < \cdots < \lambda_n$ sont des nombres réels :*

- *le système $(e^{\lambda_0 x}, \ldots, e^{\lambda_n x})$ vérifie la condition de Haar sur tout intervalle $[a, b]$ ;*
- *le système*
$$\left(\frac{1}{x-\lambda_0}, \ldots, \frac{1}{x-\lambda_n}\right)$$
*vérifie la condition de Haar sur tout intervalle $[a, b]$ ne contenant pas un des $\lambda_j$ ;*
- *le système $(1, \cos(x), \sin(x), \cos(2x), \sin(2x), \ldots, \cos(nx), \sin(nx))$ vérifie la condition de Haar sur tout intervalle $[a, b]$ tel que $0 \leq a < b < 2\pi$.*

*Le lecteur intéressé retrouvera ces exemples ainsi que d'autres dans [BE95].*

### 2.2.2 Exemples en l'absence de condition de Haar

La condition de Haar est donc assez générale et nous permet, en pratique, de faire de l'approximation polynomiale indifféremment en erreur absolue ou en erreur relative. Mais qu'en est-il lorsqu'on veut imposer la valeur d'un coefficient du polynôme ? Imaginons qu'on veuille approcher une fonction $f$ sur un intervalle $[a, b]$ par un polynôme de la forme

$$p(x) = \sum_{j=0}^{j_0-1} a_j\, x^j \;+\; \alpha\, x^{j_0} \;+\; \sum_{j=j_0+1}^{n} a_j\, x^j \quad \text{où } \alpha \text{ est une constante réelle donnée.}$$

Dans ce cas, minimiser l'erreur absolue $p - f$ ou l'erreur relative $p/f - 1$ est équivalent à minimiser $q - g$, où $g$ est une certaine fonction et $q$ est un « polynôme » écrit dans une certaine base $\varphi_0, \ldots, \varphi_{n-1}$. Les fonctions $g$ et $\varphi_j$ sont données par le tableau en figure 2.3.

| Type d'erreur | $\varphi_0(x), \ldots, \varphi_{n-1}(x)$ | $g(x)$ |
|---|---|---|
| Absolue | $(1, x, \ldots, x^{j_0-1}, x^{j_0+1}, \ldots, x^n)$ | $f(x) - \alpha\, x^{j_0}$ |
| Relative | $\dfrac{1}{f(x)}, \dfrac{x}{f(x)}, \ldots, \dfrac{x^{j_0-1}}{f(x)}, \dfrac{x^{j_0+1}}{f(x)}, \ldots, \dfrac{x^n}{f(x)}$ | $1 - \alpha\, \dfrac{x^{j_0}}{f(x)}$ |

**Figure 2.3 :** Fixer la valeur d'un coefficient revient à faire de l'approximation dans une base à trous.

De telles bases à trous ne vérifient pas nécessairement la condition de Haar. Dans cette section, nous allons en étudier les conséquences pratiques sur quelques exemples. Commençons par un résultat encourageant : si $[a, b]$ ne contient pas 0, la condition de Haar est remplie.

**Proposition 2.14.** *Soit $[a, b]$ un intervalle tel que $0 < a$, et soient $\alpha_0 < \cdots < \alpha_n$ des entiers quelconques (on peut même supposer des réels quelconques). Alors, le système $x^{\alpha_0}, \ldots, x^{\alpha_n}$ vérifie la condition de Haar.*

*Démonstration.* Cette proposition est donnée comme exercice dans [BE95]. En voici une preuve. On raisonne par récurrence sur $n$ et on démontre que la propriété (v) de la définition 2.9 est vérifiée.

Si $n = 0$, tout « polynôme » s'écrit $a_0 x^{\alpha_0}$. Puisque $[a, b]$ ne contient pas 0, si $a_0 \neq 0$, le « polynôme » n'a pas de racine dans $[a, b]$.

Supposons la propriété vraie jusqu'à $n-1$. Soient $\alpha_0 < \cdots < \alpha_n$ des nombres réels et soit $p$ défini par

$$p = \sum_{j=0}^{n} a_j\, x^{\alpha_j}, \qquad \text{où les } a_j \text{ sont des réels.}$$

On suppose que $p$ a au moins $n+1$ racines distinctes dans $[a,\, b]$ et on veut montrer que tous les $a_j$ sont nuls :

Cas n°1 : si $a_0$ est nul. Alors, $p$ s'écrit dans la base $(x^{\alpha_1}, \ldots, x^{\alpha_n})$ et l'hypothèse de récurrence peut donc s'appliquer. Comme $p$ a strictement plus que $n-1$ racines dans l'intervalle, ses coefficients dans la base $(x^{\alpha_1}, \ldots, x^{\alpha_n})$ sont tous nuls : $a_1 = \cdots = a_n = 0$.

Cas n°2 : si $a_0$ est non nul. Alors, $p$ s'écrit

$$p(x) = a_0\, x_0^{\alpha} \left(1 + \frac{a_1}{a_0} x^{\alpha_1-\alpha_0} + \cdots + \frac{a_n}{a_0} x^{\alpha_n-\alpha_0}\right) = a_0\, x_0^{\alpha}\, q(x).$$

Puisque $0 < a$, $a_0\, x^{\alpha_0}$ ne s'annule pas dans $[a,\, b]$ et le « polynôme » $q$ a donc les mêmes racines que $p$ dans $[a,\, b]$. Entre deux racines successives de $q$, $q'$ s'annule d'après le théorème de Rolle. Donc $q'$ a au moins $n$ racines distinctes dans l'intervalle. Or

$$q'(x) = b_1\, x^{\beta_1} + \cdots + b_n\, x^{\beta_n} \quad \text{avec } b_j = \frac{a_j}{a_0}\,(\alpha_j - \alpha_0) \text{ et } \beta_j = \alpha_j - \alpha_0 - 1.$$

On a $\beta_1 < \cdots < \beta_n$, donc l'hypothèse de récurrence s'applique et, comme $q'$ a strictement plus que $n-1$ racines dans $[a,\, b]$, tous les $b_j$ sont nuls. Il suit que $a_1 = \cdots = a_n = 0$, donc, en réalité, $q = 1$, ce qui contredit le fait que $q$ a au moins $n+1$ racines.

□

En général, le système $(x^{\alpha_0}, \ldots, x^{\alpha_n})$ ne vérifie pas la condition de Haar si $a = 0$. Mais lorsqu'on est dans cette situation, on peut souvent se ramener au cas classique par passage à la limite. Le théorème d'alternance, l'unicité, etc. sont alors vérifiés.

**Exemple 2.15.** *Soit $g$ une fonction continue définie sur un intervalle $[0,\, b]$ et valant 0 en 0. On cherche à approcher $g$ par un polynôme de la forme $a_1 x + \cdots + a_n x^n$. Le système $\varphi_1 = x, \ldots, \varphi_n = x^n$ ne vérifie pas la condition de Haar sur $[0,\, b]$ puisque toutes les fonctions de base s'annulent en 0. Néanmoins, il existe un unique polynôme de meilleure approximation $p^\star$ de $g$ sur $[0,\, b]$ et ce polynôme vérifie le théorème d'alternance.*

*Démonstration.* L'idée générale de la preuve est la suivante : on se place sur un intervalle $[\delta,\, b]$ (avec $\delta > 0$) sur lequel on sait qu'il y a unicité du polynôme de meilleure approximation et sur lequel le théorème d'alternance est valide. On montre que lorsqu'on fait tendre $\delta$ vers 0, la suite des polynômes de meilleure approximation correspondants tend vers un certain polynôme $p^\star$. Comme $p^\star - g$ vaut 0 en 0, $p^\star - g$ ne risque pas d'atteindre sa valeur extrême trop près de 0 et donc les points où $p^\star - g$ atteint ses valeurs extrêmes sont tous compris dans un intervalle $[\delta,\, b]$ ($\delta > 0$). Le polynôme $p^\star$ coïncide donc avec le polynôme de meilleure approximation sur $[\delta,\, b]$, lequel vérifie bien le théorème d'alternance. La figure 2.4 illustre la situation. À présent, formalisons ce raisonnement.

**Première étape :** on considère une suite $(\delta_k)$ tendant en décroissant vers 0. On note $p_k^\star$ le polynôme de meilleure approximation de $g$ sur $[\delta_k,\, b]$ et $\varepsilon_k$ l'erreur optimale correspondante.

**Deuxième étape :** le théorème 2.4 affirme qu'il existe un polynôme de meilleure approximation de $g$ sur $[0,\, b]$ (l'existence est indépendante de la condition de Haar). On note $\varepsilon$ l'erreur optimale sur $[0,\, b]$. Il est clair que la suite $(\varepsilon_k)$ est croissante et majorée par $\varepsilon$.

**Troisième étape :** puisque $(\delta_k)$ est décroissante, on a les inclusions suivantes :

$$[\delta_0,\, b] \subseteq \cdots \subseteq [\delta_k,\, b] \subseteq \cdots \subseteq [0,\, b]\,.$$

Par suite, on a les inégalités suivantes :

$$\|\cdot\|_\infty^{[\delta_0,\, b]} \leq \cdots \leq \|\cdot\|_\infty^{[\delta_k,\, b]} \leq \cdots \leq \|\cdot\|_\infty^{[0,\, b]}\,.$$

**Quatrième étape :** l'espace vectoriel engendré par la fonction $g$ et les $\varphi_j$ est de dimension finie : les normes $\|\cdot\|_\infty^{[0,\, b]}$ et $\|\cdot\|_\infty^{[\delta_0,\, b]}$ y sont donc équivalentes, ce qui signifie qu'il existe $\alpha > 0$ tel que, pour tout polynôme $p = \sum_{j=0}^{n} a_j\, \varphi_j$,

$$\|p - g\|_\infty^{[0,\, b]} \leq \alpha\, \|p - g\|_\infty^{[\delta_0,\, b]}\,.$$

**Cinquième étape :** en associant les étapes 2 à 4 on obtient pour tout $k$ :

$$\|p_k^\star - g\|_\infty^{[0,\, b]} \leq \alpha\, \|p_k^\star - g\|_\infty^{[\delta_0,\, b]} \leq \alpha\, \|p_k^\star - g\|_\infty^{[\delta_k,\, b]} = \alpha\varepsilon_k \leq \alpha\varepsilon.$$

**Sixième étape :** la suite $(p_k^\star)$ est donc bornée (pour la norme $\|\cdot\|_\infty^{[0,\, b]}$) et quitte à prendre une sous-suite, on peut donc supposer que $(p_k^\star)$ converge (en norme $\|\cdot\|_\infty^{[0,\, b]}$) vers un certain polynôme $p^\star$ (car on est en dimension finie). Remarque : bien que nous le notions $p^\star$, nous n'avons pas (encore) montré qu'il s'agissait d'un polynôme de meilleure approximation sur $[0,\, b]$.

**Septième étape :** par hypothèse $p^\star - g$ vaut 0 en 0 et est continue, donc il existe $\delta > 0$ tel que

$$\forall x \in [0,\, \delta]\,, |p^\star(x) - g(x)| \leq \varepsilon_0/3.$$

**Huitième étape :** puisque $p_k^\star$ converge vers $p^\star$, il existe $K_1$ tel que, pour $k \geq K_1$,

$$\|p^\star - p_k^\star\|_\infty^{[0,\, b]} \leq \varepsilon_0/3.$$

En particulier, $\forall x \in [0,\, \delta]\,, |p^\star(x) - p_k^\star(x)| \leq \varepsilon_0/3$.

**Neuvième étape :** puisque $(\delta_k)$ tend vers 0, il existe $K_2$ tel que, pour $k \geq K_2$, $\delta_k \leq \delta$. Soit alors $K = \max\{K_1,\, K_2\}$. On a donc

$$\forall x \in [0,\, \delta]\,, |p_K^\star(x) - g(x)| \leq 2\varepsilon_0/3 < \varepsilon_0$$

et donc *a fortiori* (étape 2) : $|p_K^\star(x) - g(x)| < \varepsilon_K$.

**Dixième étape :** par hypothèse $p_K^\star$ est le polynôme de meilleure approximation (unique) de $g$ sur $[\delta_K,\, b]$ et son erreur est $\varepsilon_K$. Nous venons donc de montrer que $\|p_K^\star - g\|_\infty^{[0,\, b]} = \varepsilon_K$. Donc l'erreur $\varepsilon$ de la meilleure approximation sur $[0,\, b]$ est plus petite que $\varepsilon_K$ ce qui (combiné avec l'étape 2) montre que $\varepsilon = \varepsilon_K$. Donc $p_K$ est en fait un polynôme de meilleure approximation sur $[0,\, b]$. Il vérifie le théorème d'alternance (puisqu'il est aussi polynôme de meilleure approximation sur $[\delta_K,\, b]$ où la condition de Haar est vérifiée).

**Onzième étape :** il n'y a qu'un seul polynôme de meilleure approximation sur $[0,\, b]$. En effet si $q^\star$ est un polynôme de meilleure approximation sur $[0,\, b]$, il vérifie

$$\|q^\star - g\|_\infty^{[0,\, b]} = \varepsilon = \varepsilon_K$$

donc *a fortiori* la norme sur $[\delta_K,\, b]$ est inférieure à $\varepsilon_K$. Or il n'y a qu'un seul polynôme fournissant une erreur inférieure ou égale à $\varepsilon_K$ sur $[\delta_K,\, b]$ : c'est $p_K^\star$. Donc $q^\star$ et $p_K^\star$ coïncident sur $[\delta_K,\, b]$. Ils sont donc tout simplement égaux. □

Jusqu'à présent, nous avons plutôt été portés à l'optimisme : la condition de Haar est souvent vérifiée, et même lorsqu'elle ne l'est pas, il semble qu'on puisse s'y ramener, peu ou prou. Notre optimisme s'arrêtera là : l'exemple suivant montre qu'en l'absence de condition de Haar, on peut perdre le théorème d'alternance et l'unicité du polynôme de meilleure approximation.

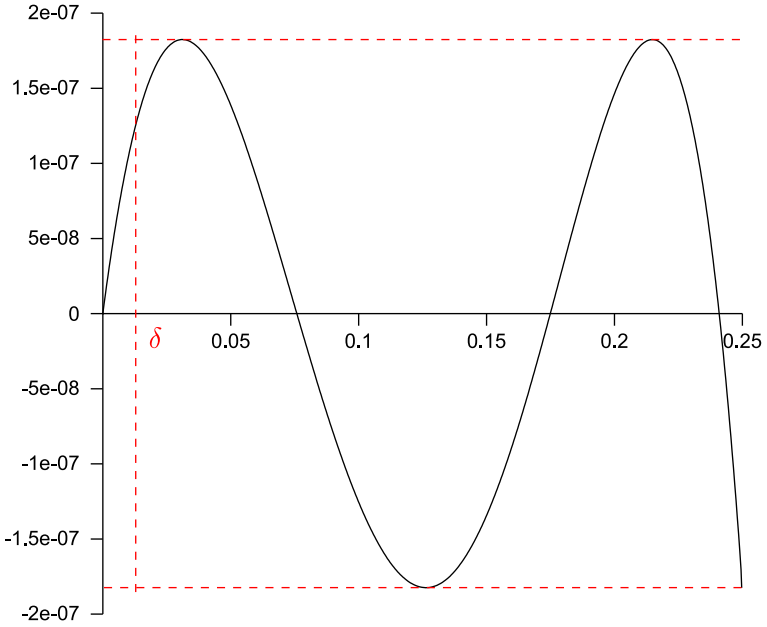



**Figure 2.4 :** Meilleure approximation de la fonction sin sur l'intervalle $[0, 1/4]$ dans la base $x$, $x^2$, $x^3$. Le polynôme de meilleure approximation sur $[\delta, 1/4]$ l'est aussi sur $[0, 1/4]$.

**Exemple 2.16.** *On cherche à approcher la fonction* exp *en erreur relative par un polynôme de la forme* $p(x) = a_2x^2 + a_4x^4 + a_6x^6$ *sur l'intervalle* $[-2^{-5}, 2^{-5}]$. *Le problème consiste donc en la minimisation de* $\|a_2\varphi_2 + a_4\varphi_4 + a_6\varphi_6 - 1\|_\infty$ *où* $\varphi_j(x) = x^j/\exp(x)$. *Ce problème admet une infinité de solutions. Certaines vérifient la condition d'alternance, d'autres non.*

*Démonstration.* Première remarque :

$$\frac{p(0)}{\exp(0)} - 1 = -1.$$

Donc la norme sup de l'erreur relative entre $p$ et exp sera supérieure ou égale à 1 pour tout polynôme $p$. Chercher $p$ sous la forme demandée est donc une très mauvaise idée si on cherche une bonne approximation. Cela dit, le propos de cet exemple n'est pas de trouver une bonne approximation, mais de montrer qu'en l'absence de condition de Haar, les théorèmes classiques deviennent faux.

Remarquons que si

$$\forall x \in \left[-2^{-5}, 0\right], \frac{p(x)}{\exp(x)} - 1 \leq 1,$$

c'est aussi vrai pour $x \in [0, 2^{-5}]$ (car $p$ est pair et, pour tout $x \geq 0$, $\exp(x) \geq \exp(-x)$). De même, puisque $p(x)/\exp(x) - 1 \geq -1$ est équivalent à $p(x) \geq 0$, il suffit que ce soit vrai pour $x \in [-2^{-5}, 0]$ pour être vrai sur tout l'intervalle.

Sur tout intervalle de la forme $[-2^{-5}, -\delta]$ (avec $\delta > 0$), la condition de Haar est réalisée et, par un argument de passage à la limite du même type que celui développé dans l'exemple 2.15, on pourrait montrer qu'il existe un polynôme $p^\star$ de meilleure approximation sur $[-2^{-5}, 0]$ qui vérifie le théorème d'alternance. D'après les remarques précédentes, ce même $p^\star$ est de meilleure approximation sur $[-2^{-5}, 2^{-5}]$ et il vérifie le théorème d'alternance. L'erreur correspondante est représentée en figure 2.5. Au passage, on notera que 1 est effectivement la meilleure erreur possible.

Cependant, $p^\star$ n'est pas le seul polynôme de meilleure approximation. Pour que $p$ soit polynôme de meilleure approximation, il faut et il suffit que

$$\forall x \in \left[-2^{-5}, 2^{-5}\right], \left|\frac{p(x)}{\exp(x)} - 1\right| \leq 1$$

ou de façon équivalente : $p(x)/\exp(x) \in [0, 2]$. Si c'est vrai de $p$, c'est *a fortiori* vrai de $\lambda p$ où $\lambda \in [0, 1]$. Voilà déjà une infinité de solutions. On notera au passage le cas extrême du polynôme nul qui est de meilleure approximation.

Plus généralement, n'importe quel polynôme $p$ qui reste compris entre 0 et $2\exp$ sur $\left[-2^{-5}, 2^{-5}\right]$ est de meilleure approximation. La figure 2.5 représente quelques courbes d'erreurs optimales. □

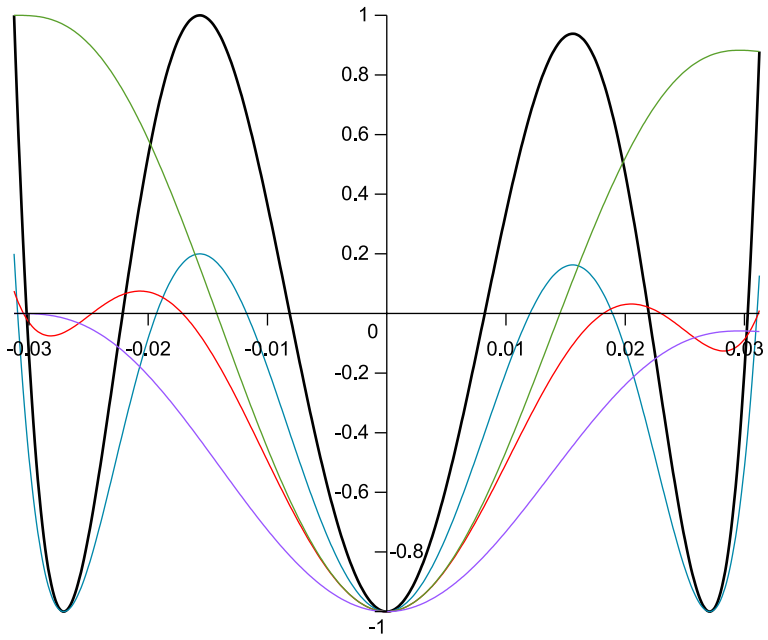



**Figure 2.5 :** Graphes d'erreur de plusieurs meilleures approximations de exp en erreur relative sur l'intervalle $\left[-2^{-5}, 2^{-5}\right]$ dans la base $x^2$, $x^4$, $x^6$. Seule l'erreur dessinée en gras vérifie la condition d'alternance.

Dans l'exemple que nous venons de traiter, un des polynômes optimaux vérifiait quand même la condition d'alternance. On pourrait donc espérer que le sens réciproque du théorème reste vrai, même en l'absence de condition de Haar : si l'erreur vérifie les conditions du théorème, le polynôme serait optimal. Là encore, il n'en est rien comme le montre l'exemple suivant :

**Exemple 2.17.** *Cet exemple provient d'une étude de cas que nous avons réellement eu l'occasion de faire lors d'une collaboration avec John Harrison (Intel Corp., Portland Oregon, USA). Ce n'est donc en rien un exemple* ad hoc. *On cherche à approcher la fonction* $f = x \mapsto \log_2(1+x)/x$ *en erreur absolue par un polynôme de degré* 11 *sur* $[-1/32, 1/32]$.
*Le polynôme qu'on voudra finalement utiliser devra avoir son coefficient d'ordre* 1 *représentable sur un* `double` *étendu, c'est-à-dire sur un nombre flottant avec* 64 *bits de précision. Il n'est pas déraisonnable de penser que ce coefficient doit être proche du coefficient correspondant dans le développement de Taylor de* $f$, *c'est-à-dire* $-1/(2\ln(2))$. *Nous posons donc* $a_1$ *le nombre flottant en* `double` *étendu le plus proche de* $-1/(2\ln(2))$. *En fait, nous détaillerons en section 2.3.2, page 104 et suivantes, une méthode basée sur la programmation linéaire et*

*qui permet d'établir rigoureusement qu'il s'agit du meilleur choix possible pour le coefficient d'ordre* 1.
*Ce coefficient étant fixé, nous cherchons donc à approcher* $(f - a_1 x)$ *par un polynôme de la forme* $p(x) = a_0 + a_2 x^2 + a_3 x^3 + \cdots + a_{11} x^{11}$. *Il existe un polynôme* $p$ *(dont les coefficients* $a_i$ *sont donnés ci-dessous) pour lequel l'erreur oscille suivant les termes du théorème d'alternance. Il fournit une erreur d'environ* $8\,e{-}23$. *Ce n'est pas le polynôme de meilleure approximation : celui-ci fournit une erreur d'environ* $6.1\,e{-}23$ *et ne respecte pas le théorème d'alternance. Les coefficients* $a_i^\star$ *correspondants sont donnés ci-dessous. En l'occurrence, il s'agit de l'unique polynôme de meilleure approximation (on remarquera donc au passage qu'en l'absence de condition de Haar, on peut quand même avoir unicité). La figure 2.6 montre les deux courbes d'erreur.*

$$
\begin{aligned}
a_0 &\simeq 1.44269504088896340735985414 & a_0^\star &\simeq 1.44269504088896340735988493\\
a_2 &\simeq 0.48089834696298780724091693 & a_2^\star &\simeq 0.48089834696298780454074864\\
a_3 &\simeq -0.36067376022224054708460799 & a_3^\star &\simeq -0.36067376022224070961803388\\
a_4 &\simeq 0.28853900817773965395612230 & a_4^\star &\simeq 0.28853900817776212011464496\\
a_5 &\simeq -0.24044917348333237718877019 & a_5^\star &\simeq -0.24044917348198569311091241\\
a_6 &\simeq 0.20609929177171215871825439 & a_6^\star &\simeq 0.20609929170743775554447682\\
a_7 &\simeq -0.18033687566734942639079274 & a_7^\star &\simeq -0.18033687939922916504429064\\
a_8 &\simeq 0.16029904848267013376922360 & a_8^\star &\simeq 0.16029912362313663048257766\\
a_9 &\simeq -0.14427408508639172657712257 & a_9^\star &\simeq -0.14426985368290754906212961\\
a_{10} &\simeq 0.13149777696669505643673217 & a_{10}^\star &\simeq 0.13146698196289650131647857\\
a_{11} &\simeq -0.11863654109336435301744638 & a_{11}^\star &\simeq -0.12032650307899117926667816
\end{aligned}
$$

Dans cet exemple, le polynôme qui vérifie la condition d'alternance fournit une erreur du même ordre de grandeur que le polynôme optimal. Ce n'est évidemment pas forcément le cas : sur certains exemples, on a pu voir jusqu'à un facteur $10^{50}$ entre l'erreur fournie par un minimax et l'erreur fournie par un polynôme vérifiant le théorème d'alternance.

On l'aura compris, dès que l'on souhaite faire de l'approximation polynomiale « à trous » (c'est-à-dire par un polynôme dont certains monômes sont nuls) sur un intervalle contenant 0, on risque de perdre la condition de Haar. La théorie classique n'est alors plus valide. On peut toujours essayer de trouver une erreur qui oscille suivant les conditions du théorème d'alternance ; parfois on en trouvera et parfois on échouera à en trouver. Quand bien même on en trouverait, ce ne serait pas nécessairement l'erreur optimale (comme le montre l'exemple précédent).

Un autre exemple : on peut souhaiter approcher autre chose que des fonctions univariées définies sur un intervalle. À vrai dire, l'ensemble de définition des fonctions importe peu. Si $X$ est un espace compact et si $g$, $\varphi_0$, …, $\varphi_n$ sont des fonctions continues de $X$ dans $\mathbb{R}$, la norme sup est bien définie et on peut se poser la question de la meilleure approximation de $g$ par une combinaison linéaire des $\varphi_j$. Le cas concret le plus directement intéressant est celui de fonctions multivariées : par exemple $g$ serait une fonction définie sur le carré $[0, 1] \times [0, 1]$ qu'on chercherait à approcher par un polynôme bivarié de degré inférieur ou égal à 3. Dans ce cas, il n'y a même plus lieu de se poser la question de savoir si le théorème d'alternance est vrai ou non : le théorème n'a tout simplement plus de sens. Atteindre les extrema de façon alternée n'a de sens que si l'ensemble de définition $X$ est muni d'un ordre naturel (ce qui est le cas des intervalles de $\mathbb{R}$, mais beaucoup moins clairement d'un carré de $\mathbb{R}^2$).

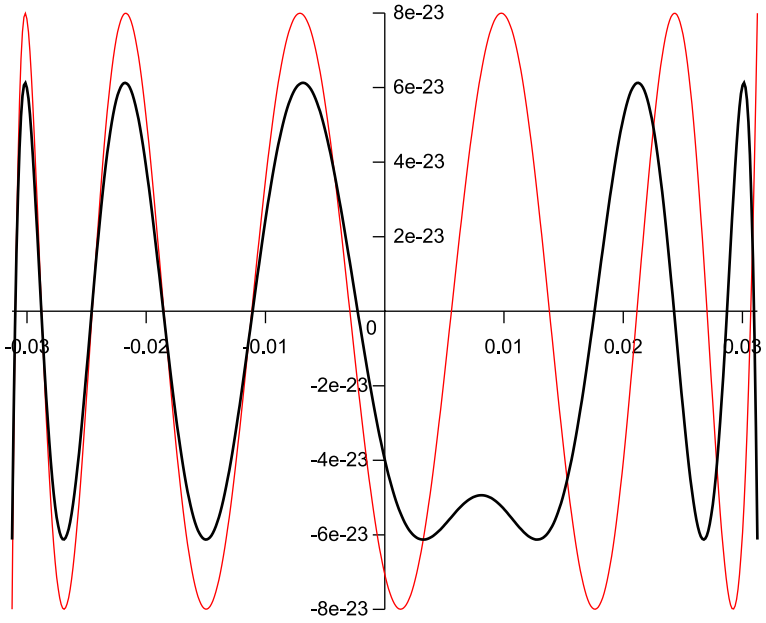



**Figure 2.6 :** Graphes d'erreur de deux polynômes d'approximation de $\log_2(1+x)/x$ en erreur absolue sur l'intervalle $[-1/32, 1/32]$. Les polynômes sont de degré 11 avec le terme de degré 1 fixé. La courbe marquée en gras correspond au polynôme optimal. Il ne vérifie pas la condition d'alternance. L'autre polynôme vérifie la condition mais n'est pas optimal.

### 2.2.3 Meilleure approximation, cadre général

Stiefel a étudié [Sti59] le problème de meilleure approximation lorsque l'ensemble $X$ est fini. Il suppose que la condition de Haar est vérifiée, c'est-à-dire que pour tout choix de $n+1$ points distincts $x_0, \ldots, x_n$ de $X$, le déterminant de la définition 2.9 est non nul.

On se donne donc $n+1$ fonctions $\varphi_0, \ldots, \varphi_n$ de $X$ dans $\mathbb{R}$ et une fonction à approcher $g$. On suppose la condition de Haar vérifiée et on cherche un « polynôme » $p = \sum_{j=0}^{n} a_j \varphi_j$ qui minimise $\mu = \max_{x\in X} |p(x) - g(x)|$. Si $X$ a $n+1$ points (ou moins), il est possible d'interpoler exactement $g$ en chaque point de $X$. Dans ce cas, $\mu = 0$ et on trouve les $a_j$ en résolvant un simple système linéaire.

Stiefel s'intéresse d'abord au cas où $X$ contient exactement $n+2$ points, c'est-à-dire le cas non trivial le plus simple. Il énonce une généralisation des théorèmes d'alternance et de La Vallée Poussin. Il étudie alors le cas où $X$ contient un nombre fini quelconque d'éléments. Il donne un algorithme, *l'algorithme d'échange*, qui calcule le « polynôme » de meilleure approximation. Le principe de l'algorithme est le suivant :

1. On choisit un sous-ensemble $X_1 \subset X$ contenant $n+2$ éléments et on calcule le « polynôme » $p_1$ de meilleure approximation sur $X_1$. On note $\mu_1$ l'erreur correspondante :
$$\mu_1 = \max_{x\in X_1} |p_1(x) - g(x)|.$$
2. On regarde alors l'erreur de $p_1$ sur $X$ tout entier :
$$\|p_1 - g\|_\infty = \max_{x\in X} |p_1(x) - g(x)|.$$
Si $\|p_1 - g\|_\infty = \mu_1$, on a trouvé le « polynôme » de meilleure approximation sur $X$.
3. Sinon, on note $x^\star$ un point où $\|p_1 - g\|_\infty$ est atteinte. On réalise alors un *échange* : on échange $x^\star$ avec l'un des points $y$ de $X_1$. On obtient ainsi un nouveau sous-ensemble $X_2 \subset X$ de $n+2$ points qui contient $x^\star$ et tous les points de $X_1$ sauf $y$.

4. Et ainsi de suite : on calcule la meilleure approximation $p_2$ sur $X_2$ et on note $\mu_2$ l'erreur correspondante, etc.

Stiefel explique comment choisir $y$ à chaque étape de telle sorte que la suite des $\mu_i$ soit strictement croissante. L'algorithme ne peut donc pas boucler et s'arrête en un nombre fini d'étapes puisqu'il n'y a qu'un nombre fini de sous-ensembles à $n+2$ éléments de $X$.

Par ailleurs, Stiefel a aussi montré [Sti60] le lien qu'il y a entre l'algorithme d'échange et l'algorithme du simplexe : le problème de meilleure approximation sur un ensemble fini $X$ peut s'exprimer comme un problème de programmation linéaire. Vu sous cet angle, l'algorithme d'échange n'est rien d'autre que l'algorithme du simplexe dual appliqué au problème de programmation linéaire. Stiefel fait remarquer que, bien que le déroulement des deux algorithmes soit formellement identique, l'algorithme d'échange est plus concis et efficace à implémenter que l'algorithme du simplexe dual.

Descloux, qui était en thèse sous la direction de Stiefel, a publié dans sa thèse [Des60] (résumée dans [Des61]), une généralisation des travaux de Stiefel au cas où la condition de Haar n'est pas remplie.

Dans le cas de l'approximation polynomiale usuelle où $X$ est un intervalle compact $[a, b]$ et où l'on cherche des polynômes dans la base des monômes $1, x, \ldots, x^n$, il existe un algorithme dû à Rémez [Rem34] pour calculer le polynôme de meilleure approximation. On distingue en fait parfois deux algorithmes [Che82] appelés *premier algorithme de Rémez* et *second algorithme de Rémez* ; c'est le second qui est intéressant en pratique et le premier est souvent ignoré de sorte qu'on parle souvent de *l'algorithme de Rémez* pour désigner le second algorithme ([Mul06] par exemple). Veidinger a prouvé [Vei60] que cet algorithme a une convergence quadratique sous des conditions raisonnables (qui sont rarement mentionnées). L'algorithme de Stiefel peut parfaitement s'appliquer au cas où $X$ est un ensemble métrique compact (pas nécessairement fini) lorsque la condition de Haar est vérifiée. Il s'agit en fait d'une généralisation de l'algorithme de Rémez. La preuve de convergence de l'algorithme de Rémez ([Che82, Chap. 3]) fonctionne encore (sans modification) lorsqu'on l'applique à l'algorithme de Stiefel.

Lorsque la condition de Haar n'est pas vérifiée et que l'ensemble $X$ n'est pas fini, les choses sont moins claires. L'algorithme de Descloux peut évidemment être appliqué mais il n'y a pas de garantie de convergence. Osborne et Watson [Wat74] ont utilisé le point de vue de la programmation linéaire pour résoudre cette question. Indépendamment, semble-t-il, Laurent et Carasso ont proposé un algorithme de type *algorithme d'échange* (voir [CL76], qui fait suite à une série d'articles des auteurs sur le sujet). Nous n'avons pas eu le temps d'étudier en détail et d'implanter ces algorithmes. À l'heure actuelle, la commande `remez` de Sollya correspond à une version modifiée de l'algorithme de Stiefel. Cet algorithme peut échouer dans certains cas mais, pour des raisons que nous verrons plus tard, les situations d'échec sont rares en pratique. Théoriquement, rien ne garantit non plus sa convergence mais nous n'avons jamais rencontré une telle situation. En pratique il est donc très satisfaisant et efficace. Cependant, à moyen terme, Sollya devrait contenir une implémentation des algorithmes génériques de Laurent/Carasso ou Watson/Osborne.

### 2.2.4 Étude du cas à $n+2$ points

L'algorithme de Stiefel repose sur la résolution du problème de meilleure approximation lorsque $X$ contient exactement $n+2$ points. Ce problème est important pour deux raisons :

— c'est le problème non trivial le plus élémentaire et, comme nous allons le voir, on est capable d'en calculer analytiquement la solution ;

— si $p^\star$ désigne la solution du problème général (où $X$ est quelconque, possiblement infini), on peut généralement trouver un sous-ensemble $X^\star \subset X$ de $n+2$ points tel que $p^\star$ soit solution du problème de meilleure approximation sur $X^\star$.

Ainsi, peu importe la façon dont on arrive à trouver le sous-ensemble $X^\star$ : une fois qu'on l'a, il suffit de résoudre le problème d'approximation sur $X^\star$ pour avoir résolu le problème sur $X$.

Supposons donc que $X = \{x_0, \ldots, x_{n+1}\}$. Notre objectif est de trouver $(a_0, \ldots, a_n) \in \mathbb{R}^{n+1}$ tel que

$$\begin{pmatrix} \varepsilon_0 \\ \vdots \\ \varepsilon_{n+1} \end{pmatrix} = \begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_n(x_0) \\ \vdots & \vdots & \vdots \\ \varphi_0(x_{n+1}) & \cdots & \varphi_n(x_{n+1}) \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} - \begin{pmatrix} g(x_0) \\ \vdots \\ g(x_{n+1}) \end{pmatrix} \tag{2.3}$$

ait la plus petite norme infini possible (la norme infini d'un vecteur étant l'analogue discret de la norme sup : $\left\| {}^t(\varepsilon_0, \ldots, \varepsilon_{n+1}) \right\|_\infty = \max\{|\varepsilon_0|, \ldots, |\varepsilon_{n+1}|\}$).

Notons que le système est rectangulaire : $n+2$ lignes par $n+1$ colonnes. Si $X$ comporte plus d'éléments, le nombre de lignes augmente, le nombre de colonnes restant $n+1$. Nous supposerons que la matrice de l'équation (2.3) est de rang $n+1$ (c'est-à-dire de rang maximal). En d'autres termes, les fonctions $\varphi_0, \ldots, \varphi_n$ forment un système libre sur $X$.

Puisqu'il y a $n+2$ lignes et $n+1$ colonnes, dans ce système de rang $n+1$, il existe une relation linéaire non triviale entre les lignes de la matrice : en notant $v_i = (\varphi_0(x_i), \ldots, \varphi_n(x_i))$,

$$\exists (\lambda_0, \ldots, \lambda_{n+1}) \in \mathbb{R}^{n+2} - \{(0, \ldots, 0)\}, \quad \text{tel que} \quad \sum_{i=0}^{n+1} \lambda_i \, v_i = 0.$$

Cette relation est unique, à un facteur multiplicatif près ; nous l'appellerons l'équation caractéristique du système.

**Remarque 2.18.** *Pour tout « polynôme » $p$, on a*

$$\sum_{i=0}^{n+1} \lambda_i \, p(x_i) = 0.$$

*En effet, posons $\alpha = {}^t(a_0, \ldots, a_n)$ le vecteur colonne formé par les coefficients de $p$ ; alors, pour tout $i$, $p(x_i) = \langle v_i, \alpha \rangle$, où $\langle \cdot, \cdot \rangle$ désigne le produit scalaire de $\mathbb{R}^{n+1}$. Par suite,*

$$\sum_{i=0}^{n+1} \lambda_i \, p(x_i) = \sum_{i=0}^{n+1} \lambda_i \, \langle v_i, \alpha \rangle = \left\langle \sum_{i=0}^{n+1} \lambda_i \, v_i, \, \alpha \right\rangle = 0.$$

Cette remarque a une conséquence importante : en notant $\varepsilon_i = p(x_i) - g(x_i)$ l'erreur commise en $x_i$, on a

$$\sum_{i=0}^{n+1} \lambda_i \, \varepsilon_i = - \sum_{i=0}^{n+1} \lambda_i \, g(x_i).$$

La valeur $-\sum_{i=0}^{n+1} \lambda_i \, g(x_i)$ est une constante indépendante de $p$ que nous noterons $C$. Quitte à multiplier tous les $\lambda_i$ par $-1$, nous pouvons la supposer positive.

**Définition 2.19** (« Polynôme » de référence)**.** Cette définition est énoncée par Stiefel [Sti59].

On dit qu'un « polynôme » est *de référence* lorsque pour tout $i$, $\varepsilon_i$ est du même signe que $\lambda_i$. Si $\lambda_i$ est nul, $\varepsilon_i$ peut avoir n'importe quelle valeur.

Remarque : si $C$ avait été négatif, on aurait demandé aux $\varepsilon_i$ d'être systématiquement de signe opposé à $\lambda_i$. Une définition alternative (qui ne dépend pas du signe de $C$) est la suivante : un « polynôme » est de référence lorsque tous les produits $\lambda_i \, \varepsilon_i$ ont le même signe (au sens large).

Lorsque $X$ est formé de $n+2$ nombres réels ordonnés $x_0 < \cdots < x_{n+1}$ et que la condition de Haar est vérifiée, on peut montrer que les $\lambda_i$ de l'équation caractéristique ont leurs signes alternés.

Un « polynôme » est donc de référence si et seulement si son signe alterne entre deux $x_i$ successifs. Lorsque la condition de Haar n'est pas réalisée ou lorsque $X$ n'est pas une partie de $\mathbb{R}$, les $\lambda_i$ n'ont aucune raison d'avoir leurs signes alternés ; la notion de « polynôme » de référence est alors la généralisation de la condition d'alternance.

**Théorème 2.20** (Solution de référence)**.** *Il existe un unique réel positif $\varepsilon$ tel qu'il existe un « polynôme » de référence pour lequel :*

— *si $\lambda_i \neq 0$, $|\varepsilon_i| = \varepsilon$ ;*

— *si $\lambda_i = 0$, $|\varepsilon_i| \leq \varepsilon$.*

*Nous appellerons un tel « polynôme » une* solution de référence.

*Démonstration.* L'énoncé et la preuve de ce théorème sont dûs à Stiefel [Sti59].

On raisonne par condition nécessaire : si un tel « polynôme » existe, il doit vérifier

$$\sum_{i=0}^{n+1} \lambda_i \, \varepsilon_i = C.$$

Comme il est de référence, $\lambda_i \, \varepsilon_i$ est positif pour tout $i$ et donc $\lambda_i \, \varepsilon_i = |\lambda_i| \, \varepsilon$. Alors $\sum_{i=0}^{n+1} |\lambda_i| \, \varepsilon = C$. Par suite, il ne peut y avoir qu'une seule valeur possible pour $\varepsilon$ et c'est

$$\varepsilon = \frac{C}{\sum\limits_{i=0}^{n+1} |\lambda_i|}.$$

Soit $i$ tel que $\lambda_i \neq 0$. Alors $p(x_i)$ doit vérifier $p(x_i) = c_i$, où $c_i = g(x_i) + \operatorname{sgn}(\lambda_i) \, \varepsilon$ (on note $\operatorname{sgn}(\lambda_i) = \pm 1$ le signe de $\lambda_i$). Pour les $i$ tels que $\lambda_i = 0$, on peut choisir une valeur arbitraire $\varepsilon_i \in [-\varepsilon, \varepsilon]$ et on pose $c_i = g(x_i) + \varepsilon_i$.

Il ne nous reste qu'à montrer l'existence d'un « polynôme » $p$ vérifiant ces contraintes et nous aurons fini. Puisque la matrice de l'équation (2.3) est de rang $n+1$, il existe un indice $i_0$ tel que la matrice carrée obtenue en supprimant la ligne correspondante soit inversible. Alors, le système suivant admet une unique solution $\alpha = {}^t(a_0, \ldots, a_n)$ :

$$\begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_n(x_0) \\ \vdots & \vdots & \vdots \\ \varphi_0(x_{i_0-1}) & \cdots & \varphi_n(x_{i_0-1}) \\ \varphi_0(x_{i_0+1}) & \cdots & \varphi_n(x_{i_0+1}) \\ \vdots & \vdots & \vdots \\ \varphi_0(x_{n+1}) & \cdots & \varphi_n(x_{n+1}) \end{pmatrix} \begin{pmatrix} a_0 \\ \\ \vdots \\ \vdots \\ \\ a_n \end{pmatrix} = \begin{pmatrix} c_0 \\ \vdots \\ c_{i_0-1} \\ c_{i_0+1} \\ \vdots \\ c_{n+1} \end{pmatrix}. \tag{2.4}$$

Il suffit de montrer que l'équation portant sur $i_0$ est elle aussi vérifiée : $\langle v_{i_0}, \alpha \rangle = c_{i_0}$. Un simple calcul en utilisant l'équation caractéristique, la définition de $\varepsilon$ et la définition des $c_i$ permet de conclure (on remarquera que, pour tout $i$, $\lambda_i \, c_i = \lambda_i \, (g(x_i) + \operatorname{sgn}(\lambda_i) \, \varepsilon)$ : c'est vrai par définition si $\lambda_i \neq 0$ et c'est vrai de fait si $\lambda_i = 0$). □

**Remarque 2.21.** *En utilisant les définitions de $\varepsilon$ et de $C$, on voit facilement, que pour tout « polynôme »,*

$$\varepsilon = \sum_{i=0}^{n+1} \frac{\lambda_i}{\sum\limits_{i=0}^{n+1} |\lambda_i|} \varepsilon_i.$$

*Par conséquent, $\varepsilon \leq \sum\limits_{i=0}^{n+1} \frac{|\lambda_i|}{\sum_{i=0}^{n+1} |\lambda_i|} |\varepsilon_i|$ et en particulier $\varepsilon \leq \max\limits_{i \in [\![0, n+1]\!]} |\varepsilon_i|$.*

**Proposition 2.22.** *Un « polynôme » est de meilleure approximation sur $X = \{x_0, \ldots, x_{n+1}\}$ si et seulement s'il est solution de référence pour $X$.*

*Démonstration.* L'énoncé et la preuve de cette proposition sont dûs à Stiefel [Sti59].

D'après la remarque précédente, pour tout « polynôme », la norme de l'erreur est supérieure ou égale à $\varepsilon$. Puisque les solutions de référence ont une erreur égale à $\varepsilon$, ce sont des meilleures approximations.

Réciproquement, soit $p$ un « polynôme » de meilleure approximation ; la norme de son erreur est donc $\varepsilon$. Notons $\varepsilon_i = p(x_i) - g(x_i)$. On a donc $|\varepsilon_i| \leq \varepsilon$ et il existe une solution de référence $p^\star$ telle que $\varepsilon_i^* = \varepsilon_i$ pour tous les $i$ tels que $\lambda_i = 0$.

Soit à présent $i$ un indice quelconque. Si $\lambda_i = 0$, on a $\varepsilon_i = \varepsilon_i^\star$ et, si $\lambda_i \neq 0$, $|\varepsilon_i^\star| = \varepsilon$ donc $|\varepsilon_i| \leq |\varepsilon_i^\star|$. Il suit que $\varepsilon_i^\star$ et $\varepsilon_i^\star - \varepsilon_i$ ont le même signe (qui est aussi le signe de $\lambda_i$ puisque $p^\star$ est de référence) donc, pour tout $i$,

$$|\varepsilon_i^\star - \varepsilon_i| \cdot |\lambda_i| = (\varepsilon_i^\star - \varepsilon_i)\, \lambda_i.$$

Par suite

$$\sum_{i=0}^{n+1} |\lambda_i| \cdot |\varepsilon_i^\star - \varepsilon_i| = \sum_{i=0}^{n+1} \lambda_i\, (\varepsilon_i^\star - \varepsilon_i) = \left(\sum_{i=0}^{n+1} \lambda_i\, \varepsilon_i^\star\right) - \left(\sum_{i=0}^{n+1} \lambda_i\, \varepsilon_i\right) = C - C = 0.$$

Le membre de gauche de cette égalité est une somme de terme positifs ; ils sont donc tous nuls. On en déduit que, pour tout $i$, $\varepsilon_i = \varepsilon_i^\star$. Puisque le système de l'équation (2.4) est inversible, il est clair que deux « polynômes » qui coïncident sur les points $x_0, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_{n+1}$ ont leurs coefficients égaux. Donc $p = p^\star$, ce qui conclut la preuve. □

> **Remarque 2.23.** *Considérons le cas d'un ensemble $X$ quelconque (possiblement infini).* ***On ne suppose pas que la condition de Haar est remplie.*** *On choisit un sous-ensemble $\{x_0, \ldots, x_{n+1}\}$ de $n+2$ points distincts dans $X$. On suppose simplement que la matrice de l'équation (2.3) associée est de rang $n+1$. Notons $p$ une meilleure approximation sur ces $n+2$ points et $\varepsilon = \max_{i \in [\![0,\, n+1]\!]} |p(x_i) - g(x_i)|$. Considérons un « polynôme » $p^\star$ de meilleure approximation sur $X$ et $\mu$ l'erreur optimale correspondante $\mu = \|p^\star - g\|_\infty^X$. Alors*
>
> $$\varepsilon \leq \mu \leq \|p - g\|_\infty^X.$$
>
> *L'inégalité de droite vient de l'optimalité de $\mu$ ; si l'inégalité de gauche était fausse, on aurait $\mu < \varepsilon$ et donc $|p^\star(x_i) - g(x_i)| \leq \mu < \varepsilon$ pour tout $i \in [\![0,\, n+1]\!]$, ce qui contredirait l'optimalité de $\varepsilon$. Ce résultat est la généralisation du théorème de La Vallée Poussin (théorème 2.7 page 83).*

Ce qui est intéressant dans cette remarque, c'est que nous n'avons pas besoin de la condition de Haar. Ce résultat est très important en pratique puisque le théorème de La Vallée Poussin nous permet de mesurer à quel point un « polynôme » est proche de la meilleure approximation, sans avoir besoin de la connaître explicitement (voir l'équation (2.2) page 84).

### 2.2.5 Algorithme de Rémez

Décrivons tout d'abord l'algorithme de Rémez, qui est l'algorithme classique utilisé pour calculer le polynôme de meilleure approximation de degré inférieur ou égal à $n$ d'une fonction sur un intervalle compact $[a, b]$. Après avoir compris ce qui se passe dans ce cas simple, nous pourrons décrire l'algorithme d'échange de Stiefel.

L'algorithme de Rémez prend pour entrées une fonction continue $g$, un intervalle $[a, b]$, un entier $n$ et un paramètre réel strictement positif qualite. Il calcule un polynôme $p$ de degré inférieur ou égal à $n$ qui approche le polynôme $p^\star$ de meilleure approximation de $g$ sur $[a, b]$. Plus précisément, il utilise le théorème de La Vallée Poussin (cf. l'équation (2.2)) pour garantir que

$$\|p^\star - g\|_\infty \leq \|p - g\|_\infty \leq \|p^\star - g\|_\infty (1 + \delta), \quad \text{avec } \delta \leq \text{qualite}.$$

L'algorithme est résumé en figure « algorithme 2.1 ».

**Input** : $g$, $[a, b]$, $n$, qualite
**Output** : un polynôme $p$ tel que $\|p^\star - g\|_\infty \leq \|p - g\|_\infty \leq \|p^\star - g\|_\infty (1 + \text{qualite})$

1 Choisir $n + 2$ points $x_0 < \cdots < x_{n+1}$ dans $[a, b]$ ;
2 $\delta \leftarrow +\infty$ ;
3 **while** ($\delta >$ qualite) **do**
4 Calculer la meilleure approximation $p$ de $g$ sur $\{x_0, \ldots, x_{n+1}\}$ ;
5 $\varepsilon \leftarrow \max_{i \in [\![0, n+1]\!]} |p(x_i) - g(x_i)|$ ;
6 $M \leftarrow \|p - g\|_\infty$ ;
7 Choisir $n + 2$ nouveaux points $x_0 < \cdots < x_{n+1}$ tels que

$$\begin{cases} \exists i_0 \in [\![0, n+1]\!], |p(x_{i_0}) - g(x_{i_0})| = M \\ \forall i \in [\![0, n+1]\!], |p(x_i) - g(x_i)| \geq \varepsilon \\ \text{le signe de } p - g \text{ alterne entre deux } x_i \text{ successifs} ; \end{cases}$$

8 $\varepsilon' \leftarrow \min_{i \in [\![0, n+1]\!]} |p(x_i) - g(x_i)|\}$ ;
9 $\delta \leftarrow (M - \varepsilon')/\varepsilon'$ ;
10 **end**
11 **return** $p$;

**Algorithme 2.1 :** Algorithme de Rémez

L'algorithme est présenté sous un formalisme assez général qui permettra de l'étendre facilement au cas de l'approximation sur un ensemble $X$ quelconque. Cependant, si l'on veut implémenter spécifiquement l'algorithme pour l'approximation polynomiale sur un intervalle compact $[a, b]$, on peut effectuer l'étape 4 efficacement à l'aide d'un algorithme rapide d'interpolation (voir [Sti64]). Pour faire cette interpolation rapidement et de façon numériquement stable, on a même intérêt à représenter les polynômes dans une base bien adaptée au calcul [BT04, PT08]. L'étape 5 est séparée de l'étape 4 dans l'algorithme mais, en pratique, $p$ et $\varepsilon$ sont calculés simultanément ($\varepsilon$ est l'erreur optimale du problème d'approximation restreint à $\{x_0, \ldots, x_{n+1}\}$).

L'étape 7 laisse une certaine liberté pour le choix des nouveaux points. Là encore, la formulation est délibérément assez générale mais, en pratique, certains choix de points sont plus pertinents que d'autres. Si l'erreur $p - g$ a exactement $n + 2$ extrema locaux alternés dans l'intervalle $[a, b]$, on a intérêt à choisir les extrema locaux comme nouveau choix de points. Sous cette hypothèse (et en supposant de plus que $g$ est deux fois différentiable sur $[a, b]$), Veidinger [Vei60] a montré que la convergence de l'algorithme est quadratique, c'est-à-dire que $\delta$ est au moins élevé au carré entre deux étapes. S'il y a plus que $n + 2$ extrema locaux, le choix des nouveaux points est plus délicat ; on peut suivre par exemple la règle donnée dans le livre de Cheney [Che82, Chap. 3].

À l'étape 9, on utilise le théorème de La Vallée Poussin pour calculer un majorant $\delta$ de la qualité d'approximation de $p$. L'algorithme boucle jusqu'à ce que cette qualité soit suffisante.

Pour choisir le jeu de points initial $x_0 < \cdots < x_{n+1}$, il est souvent judicieux de considérer les points correspondant aux extrema du polynôme de Tchebychev de degré $n+1$ sur $[a, b]$ :

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{(n+1-i)\,\pi}{n+1}\right) \qquad i = 0, \ldots, n+1. \tag{2.5}$$

Si la condition de Haar est vérifiée, l'algorithme de Rémez fonctionne à l'identique dans le cas de l'approximation de $g$ sur $[a, b]$ par un « polynôme » dans une base de fonctions continues quelconques $\varphi_0, \ldots, \varphi_n$. Le théorème de Veidinger est encore valable, à la condition additionnelle que les $\varphi_j$ soient deux fois dérivables sur $[a, b]$.

### 2.2.6 Algorithme d'échange de Stiefel

Nous en arrivons à la description de l'algorithme d'échange de Stiefel. Nous considérons à nouveau un ensemble compact $X$ quelconque (pas forcément un intervalle compact de $\mathbb{R}$). Stiefel suppose que $X$ est fini et que la condition de Haar est vérifiée : pour tout choix de $n+1$ points distincts $x_0, \ldots, x_n$ dans $X$, le système

$$\begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_n(x_0) \\ \vdots & \vdots & \vdots \\ \varphi_0(x_n) & \cdots & \varphi_n(x_n) \end{pmatrix}$$

est inversible. Cette hypothèse implique que, lorsqu'on choisit $n+2$ points distincts $x_0, \ldots, x_{n+1}$ dans $X$, les $\lambda_i$ de l'équation caractéristique associée sont tous non nuls.

**Lemme 2.24** (d'échange)**.** *On suppose que la condition de Haar est vérifiée. Soient $x_0, \ldots, x_{n+1}$ des points distincts de $X$. On note $\lambda_i$ les coefficients de l'équation caractéristique correspondante. Soit $p$ un « polynôme » de référence pour ces points ; on note $\varepsilon_i = p(x_i) - g(x_i)$.*

*On considère un point $\bar{x}$ de $X$ distinct des précédents. Alors, il existe un indice $i_0$ tel que, lorsqu'on remplace $x_{i_0}$ par $\bar{x}$, $p$ soit encore de référence pour le nouveau système*

$$(x_0, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_{n+1}, \bar{x}).$$

*Démonstration.* L'énoncé et la preuve de ce lemme sont dûs à Stiefel [Sti59].

On note $v_i = (\varphi_0(x_i), \ldots, \varphi_n(x_i))$ et $\bar{v} = (\varphi_0(\bar{x}), \ldots, \varphi_n(\bar{x}))$. De plus, on note $\bar{\varepsilon} = p(\bar{x}) - g(\bar{x})$. Sans même supposer que la condition de Haar est remplie, si le système formé de $(v_0, \ldots, v_{n+1})$ est de rang $n+1$ (hypothèse que nous avions faite jusqu'à présent), on peut écrire $\bar{v}$ comme une combinaison linéaire des $v_i$ :

$$\bar{v} = \sum_{i=0}^{n+1} \mu_i\, v_i.$$

Il est facile de vérifier que le système $(x_0, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_{n+1}, \bar{x})$ a pour équation caractéristique :

$$-\lambda_{i_0}\, \bar{v} + \sum_{\substack{i=0 \\ i \neq i_0}}^{n+1} (\mu_i\, \lambda_{i_0} - \mu_{i_0}\, \lambda_i)\, v_i = 0.$$

Pour que $p$ soit de référence dans ce nouveau système, il faut et il suffit qu'il existe $s = \pm 1$ tel que

$$\begin{cases} \forall i \neq i_0,\ \operatorname{sgn}(\mu_i\, \lambda_{i_0} - \mu_{i_0}\, \lambda_i)\, \operatorname{sgn}(\varepsilon_i) = s \\ -\operatorname{sgn}(\lambda_{i_0})\, \operatorname{sgn}(\bar{\varepsilon}) = s. \end{cases}$$

Or, en divisant par $\lambda_i$ et $\lambda_{i_0}$ (ce qui est légitime lorsque la condition de Haar est vérifiée, puisqu'ils sont alors non nuls) :

$$\operatorname{sgn}(\mu_i\,\lambda_{i_0} - \mu_{i_0}\,\lambda_i) = \operatorname{sgn}(\lambda_i)\,\operatorname{sgn}(\lambda_{i_0})\,\operatorname{sgn}\left(\frac{\mu_i}{\lambda_i} - \frac{\mu_{i_0}}{\lambda_{i_0}}\right).$$

Puisque $p$ est de référence pour $(x_0, \ldots, x_{n+1})$, on a $\operatorname{sgn}(\varepsilon_i) = \operatorname{sgn}(\lambda_i)$ et donc $p$ est de référence pour le nouveau système si et seulement si pour tout $i \neq i_0$, $\operatorname{sgn}\left(\frac{\mu_i}{\lambda_i} - \frac{\mu_{i_0}}{\lambda_{i_0}}\right) = -\operatorname{sgn}(\bar{\varepsilon})$.

Suivant le signe de $\bar{\varepsilon}$, on choisira donc $i_0$ de façon à maximiser ou minimiser le rapport $\mu_{i_0}/\lambda_{i_0}$. □

L'algorithme de Stiefel est alors le suivant : on part d'un ensemble de $n+2$ points distincts quelconque $x_0, \ldots, x_{n+1}$, on calcule le « polynôme » $p$ de meilleure approximation sur ces $n+2$ points et son erreur $\varepsilon$ correspondante :

$$\varepsilon = \max_{i \in [\![0, n+1]\!]} |p(x_i) - g(x_i)|.$$

Considérons à présent la norme de $p - g$ sur l'ensemble $X$ tout entier (et non plus seulement sur l'ensemble $\{x_0, \ldots, x_{n+1}\}$) :

$$\|p - g\|_\infty^X = \max_{x \in X} |p(x) - g(x)| \geq \varepsilon.$$

Si $\|p - g\|_\infty^X = \varepsilon$, c'est que $p$ est de meilleure approximation sur $X$ tout entier et on a donc fini.

Dans le cas contraire, considérons un point $\bar{x} \in X$ pour lequel $\|p - g\|_\infty^X$ est atteint. On utilise le lemme d'échange pour échanger un des $x_i$ avec $\bar{x}$ ; on obtient ainsi un nouveau jeu de points $\{x'_0, \ldots, x'_{n+1}\}$, on calcule le « polynôme » de meilleure approximation $p'$ relatif à ces $n+2$ points, ainsi que l'erreur

$$\varepsilon' = \max_{i \in [\![0, n+1]\!]} |p'(x'_i) - g(x'_i)|,$$

et ainsi de suite.

Stiefel a montré [Sti59] que, entre deux étapes successives, on a toujours $\varepsilon < \varepsilon'$. Par conséquent, au cours de l'algorithme, on ne retombe jamais deux fois sur le même ensemble de $n+2$ points $\{x_0, \ldots, x_{n+1}\}$. Dans le cas où $X$ est fini, l'algorithme s'arrête donc nécessairement en un nombre fini d'étapes.

Descloux [Des60] a montré qu'on pouvait modifier l'algorithme d'échange pour qu'il fonctionne même si la condition de Haar n'est pas vérifiée. Dans ce cas, il arrive parfois que $\varepsilon = \varepsilon'$. Lorsque cela se produit, on dit que l'échange était *statique*. Descloux prouve qu'il n'y a pas de suite infinie d'échanges statiques. Donc là encore, si $X$ est fini, l'algorithme s'arrête en un temps fini.

Dans le cas où $X$ est infini, il est bien sûr possible d'utiliser l'algorithme d'échange, mais l'argument de finitude n'est plus valable. Si $X$ est un espace métrique compact et que la condition de Haar est vérifiée, la preuve de convergence de l'algorithme de Rémez [Che82, Chap. 3] s'applique (sans modification) : on modifie simplement la condition d'alternance de l'étape 7 (voir algorithme 2.1 page 98) par la condition *$p$ reste « polynôme » de référence pour le nouveau choix de points.* À cette même étape, le nouveau jeu de points est obtenu en utilisant l'algorithme d'échange (fourni par la preuve du lemme 2.24).

### 2.2.7 Implantation dans Sollya

Une partie des algorithmes présentés dans cette section est disponible dans Sollya via la commande `remez`. Cette commande prend en argument une fonction $g$, une liste d'entiers $(\alpha_0, \ldots, \alpha_n)$

et un intervalle $[a, b]$. Il y a deux autres paramètres, qui sont facultatifs : une fonction $w$ (par défaut, $w$ est la fonction constante à 1) et une valeur positive qualite (dans l'implémentation actuelle, la valeur par défaut est $1\,\mathrm{e}{-5}$).

La commande cherche un « polynôme » $p$ dans la base $(\varphi_0, \ldots, \varphi_n)$, où $\varphi_j(x) = w(x)\,x^{\alpha_j}$ pour approcher $g$ sur $[a, b]$. Le rôle du paramètre qualite est le même que dans l'algorithme 2.1. En jouant sur la liste des $\alpha_j$ et les fonctions $w$ et $g$, il est possible de formuler tous les problèmes d'approximation polynomiale évoqués depuis le début de ce chapitre (approximation polynomiale classique ou approximation en fixant la valeur de certains coefficients, que ce soit en erreur absolue ou relative).

L'algorithme utilisé actuellement est l'algorithme 2.1, où le choix du nouveau jeu de points à l'étape 7 est un mélange heuristique entre l'algorithme d'échange de Stiefel (où un seul point est changé à chaque étape) et l'algorithme de Rémez usuel (où on choisit le nouveau jeu de points parmi les extrema locaux de $p - g$).

Lorsque la condition de Haar est vérifiée, la convergence est assurée. Dans le cas contraire, l'algorithme est uniquement heuristique mais il est très satisfaisant en pratique. Pour que l'algorithme d'échange de Stiefel puisse s'exécuter, il suffit en pratique que les $\lambda_i$ qui apparaissent au cours de l'exécution ne soient jamais nuls. Lorsque la condition de Haar n'est pas remplie, on sait qu'il existe des choix de points $x_0, \ldots, x_{n+1}$ dont l'équation caractéristique fait apparaître des $\lambda_i$ nuls. Cependant, en règle générale, une faible perturbation sur les $x_i$ perturbe aussi les $\lambda_i$ de sorte qu'ils sont proches de 0 mais non nuls. En pratique, les $x_i$ sont obtenus, à chaque étape, par un processus numérique approché (recherche numérique des extrema locaux de $p - g$) et il est donc improbable que des $\lambda_i$ nuls apparaissent pendant l'exécution de l'algorithme. Il peut donc s'exécuter sans incident. On n'a aucune garantie de convergence, mais si l'algorithme s'arrête, on sait qu'on a effectivement trouvé un « polynôme » dont la qualité d'approximation est au moins qualite : la remarque 2.23 le garantit même lorsque la condition de Haar n'est pas vérifiée.

En l'état actuel, la commande `remez` de Sollya est satisfaisante mais de nombreuses améliorations sont encore envisageables :

— pour l'instant, la syntaxe de la commande ne permet de travailler qu'avec des bases de la forme $(x^{\alpha_0}, \ldots, x^{\alpha_n})$ ou $(x^{\alpha_0}/f(x), \ldots, x^{\alpha_n}/f(x))$, avec des $\alpha_j$ entiers. La raison de cette limitation est historique (ce sont les bases dont on se sert en pratique pour l'implantation de fonctions). Dans une prochaine version, il devrait être possible de chercher des combinaisons linéaires de fonctions $\varphi_j$ continues quelconques ;

— de la même façon, la syntaxe actuelle permet d'approcher une fonction uniquement sur un intervalle fermé $[a, b]$. Dans une prochaine version, l'utilisateur devrait pouvoir fournir une liste d'intervalles $[a_1, b_1], \ldots, [a_k, b_k]$ et demander le « polynôme » de meilleure approximation sur le compact $X = [a_1, b_1] \cup \cdots \cup [a_k, b_k]$ ;

— comme toutes les commandes de Sollya, la commande `remez` adapte la précision au cours des calculs, de façon à fournir un résultat fiable. Cependant, cette adaptation de précision n'est pas effectuée à toutes les étapes de l'algorithme, ce qui peut conduire la commande à échouer sur des instances particulièrement difficiles. Dans ce cas, il suffit d'augmenter manuellement la précision de Sollya (contrôlée par la variable globale `prec`). L'adaptation complètement automatique de la précision au cours de l'algorithme est un problème intéressant qui ne semble pas avoir été étudié jusqu'à maintenant ;

— enfin, l'implémentation actuelle a été obtenue en modifiant successivement plusieurs fois l'implémentation originale (qui ne fonctionnait qu'en présence de la condition de Haar). En conséquence, la commande est parfois plus lente qu'elle ne devrait, en particulier lorsque la fonction est difficile à approcher ou que la condition de Haar n'est pas remplie. Cette relative lenteur n'est pas un problème en soi : elle ne survient que sur des cas très particuliers, et le résultat est néanmoins obtenu en quelques dizaines de secondes. Une récriture complète de

l'algorithme devrait régler ce problème.

## 2.3 Approximation polynomiale sous contrainte discrète

Jusqu'à présent, nous nous sommes intéressés à l'approximation de fonctions par des polynômes à coefficients réels. Nous avons étudié ce qui se passait lorsqu'on avait pour contrainte de fixer la valeur de certains coefficients. Nous allons à présent nous intéresser à un autre genre de contraintes liées à la représentation des coefficients.

### 2.3.1 Modélisation du problème

En effet, la représentation des nombres a une incidence pratique sur notre problème d'approximation polynomiale. Nous ne cherchons pas à approcher une fonction $f$ par un polynôme $p$ quelconque : il faut que nous puissions représenter $p$ et calculer avec.

En pratique, les coefficients de $p$ sont stockés en mémoire dans un format numérique : le plus souvent, ce sont des nombres en virgule fixe ou en virgule flottante (voir section 1.1.1 page 34) ; dans le cadre du développement d'une libm, on peut même vouloir utiliser des expansions de nombres flottants (voir section 2.1.2 page 79). Dans la suite du chapitre, on parlera parfois de *nombre machine* pour désigner un nombre représentable dans l'un de ces formats.

Bien sûr, on n'est pas obligé d'utiliser le même format pour chaque coefficient. Prenons l'exemple (relativement générique) de l'approximation d'une fonction $f$ par un polynôme de degré 6 écrit dans la base des monômes usuels 1, $x$, $x^2$, ... sur un intervalle relativement petit et centré en 0 (par exemple, $[-1/32, 1/32]$) :

$$p(x) = \sum_{i=0}^{6} a_i x^i \simeq f(x).$$

Sur cet intervalle, $x^5$ et $x^6$ restent très petits par rapport à 1 ou $x$ ; l'influence des coefficients $a_5$ et $a_6$ sur le comportement de $p$ est donc moindre que celle des coefficients $a_0$ et $a_1$. Il est raisonnable de représenter $a_5$ et $a_6$ sur un format ayant moins de précision que celui utilisé pour $a_0$ et $a_1$. On pourra par exemple utiliser des nombres en simple précision pour $a_5$ et $a_6$ et des nombres en double précision (voire des expansions de `double` si nécessaire) pour $a_0$ et $a_1$.

Notons l'importance de la base dans laquelle sont écrits les polynômes. Lorsqu'on considère des polynômes à coefficients réels on peut travailler dans la base des monômes 1, $x$, $x^2$, $x^3$ aussi bien que, par exemple, dans la base des polynômes de Tchebychev 1, $x$, $2x^2-1$, $4x^3-3x$. Les deux bases engendrent le même ensemble constitué des polynômes de degré inférieur ou égal à 3. Ce n'est plus vrai lorsqu'on s'intéresse aux polynômes à coefficients dans un certain format. Par exemple

$$(1 - 1/2^{60}) + x^2/2^{59} = 1 + (2x^2 - 1)/2^{60}$$

est représentable avec des coefficients au format `double` dans la base des polynômes de Tchebychev, mais il ne l'est pas dans la base des monômes.

En toute généralité, notre problème se pose donc de la façon suivante :

**Problème 2.25** (Formalisation du problème)**.** *Comme d'habitude, on se donne un intervalle $[a, b]$, une fonction continue $g$ et une famille de $n+1$ fonctions $\varphi_0, \ldots, \varphi_n$ continues sur $[a, b]$ et linéairement indépendantes. Pour se faire une idée, on peut imaginer qu'il s'agit des monômes ; on peut tout aussi bien prendre une autre base des polynômes (polynômes de Tchebychev par exemple), une base « à trous » (constituée de certains monômes mais pas tous), etc. Mais rien n'oblige à se limiter aux polynômes.*

*Par ailleurs, on se donne, pour chaque fonction $\varphi_j$, un format de représentation du coefficient correspondant. Nous cherchons les « polynômes » (c'est-à-dire plus exactement des combinaisons*

*linéaires des fonctions de base $\varphi_j$) dont les coefficients vérifient les contraintes de format et dont l'erreur par rapport à $g$ en norme sup est minimale.*

On pourrait penser que ce problème n'a pas de réelle pertinence, dans la mesure où les nombres flottants constituent une approximation des réels avec une précision généralement considérée comme satisfaisante. Ainsi, on pourrait croire que le modèle discret que constituent les polynômes à coefficients représentables en machine est suffisamment proche du modèle réel : on pourrait se contenter de chercher le polynôme de meilleure approximation à coefficients réels $p^\star$ et arrondir chacun de ses coefficients au nombre représentable le plus proche. On obtient ainsi un polynôme $\widehat{p}$ dont les coefficients sont représentables en machine. Cette stratégie peut conduire à une perte importante de la qualité d'approximation. Les quatre exemples suivants illustrent le fait que $\widehat{p}$ peut fournir une qualité d'approximation largement sous-optimale.

Le tableau 2.7 montre, pour les quatre exemples ci-dessous :

— l'erreur du polynôme $p^\star$ de meilleure approximation *à coefficients réels* ;
— l'erreur du polynôme $\widehat{p}$ obtenu en arrondissant les coefficients de $p^\star$ au plus près ;
— l'erreur d'un polynôme $p$ obtenu par une des méthodes que nous allons présenter dans la suite de cette section. Les coefficients de $p$, comme ceux de $\widehat{p}$ sont représentables aux formats demandés. Cependant, comme l'illustre le tableau, $p$ peut être nettement meilleur que $\widehat{p}$.

**Exemple 2.26.** *On veut approcher la fonction* $\log_2(1+2^{-x})$ *en erreur absolue sur* $[0, 1]$ *par un polynôme de degré* 6 *dont les coefficients sont en simple précision (*24 *bits de précision).*

**Exemple 2.27.** *On veut approcher la fonction* $\cos(\pi\, x)$ *en erreur relative sur* $[0, 1/256]$ *par un polynôme pair de degré* 6 *dont les coefficients sont des* `double` *(*53 *bits de précision).*

**Exemple 2.28.** *On veut approcher la fonction* $\log_2(1 + x)$ *en erreur relative sur* $[-1/512, 1/512]$ *par un polynôme de la forme* $a_1x + \cdots + a_{13}x^{13}$. *Les coefficients* $a_1$ *et* $a_2$ *doivent être des triples-*`double`*, les coefficients* $a_3$ *à* $a_7$ *des doubles-*`double` *et tous les autres des* `double`.

**Exemple 2.29.** *On veut approcher la fonction* $(2^x-1)/x$ *en erreur absolue sur* $[-1/16, 1/16]$ *par un polynôme de degré* 9. *Le coefficient constant doit être un double-*`double`*, et tous les autres des* `double` *étendus (*64 *bits de précision).*

| Exemple | Erreur de $p^\star$ | Erreur de $\widehat{p}$ | Erreur de $p$ |
|---|---|---|---|
| 2.26 | $8,3\,\mathrm{e}{-10}$ | $119\,\mathrm{e}{-10}$ | $10,07\,\mathrm{e}{-10}$ |
| 2.27 | $1,0\,\mathrm{e}{-22}$ | $16\,\mathrm{e}{-22}$ | $3,4\,\mathrm{e}{-22}$ |
| 2.28 | $1,05\,\mathrm{e}{-40}$ | $8\,485\,\mathrm{e}{-40}$ | $4,6\,\mathrm{e}{-40}$ |
| 2.29 | $7,9\,\mathrm{e}{-25}$ | $4\,035\,\mathrm{e}{-25}$ | $445\,\mathrm{e}{-25}$ |

**Figure 2.7 :** Qualité d'approximation du polynôme $p^\star$ de meilleure approximation à coefficients réels, du polynôme arrondi $\widehat{p}$, et d'un polynôme $p$ à coefficients représentables obtenu par une des méthodes décrites dans ce chapitre.

### 2.3.2 Utilisation de la programmation linéaire

De la même façon que pour la résolution du problème à coefficients réels, la programmation linéaire apporte un point de vue intéressant et des informations précieuses. Considérons une instance du problème discret : nous avons donc un intervalle $[a, b]$, une fonction $g$, une base de « monômes » $\varphi_0, \ldots, \varphi_n$ et une liste de formats correspondants. On pose $p^\star$ un « polynôme » optimal, parmi les polynômes *à coefficients réels* et $\varepsilon^\star$ la norme sup de l'erreur. En imposant un format spécifique pour chaque coefficient, nous restreignons l'espace de recherche par rapport au problème continu. La valeur $\varepsilon^\star$ constitue donc un minorant pour le problème discret : le meilleur « polynôme » à coefficients représentables a une erreur plus grande que $\varepsilon^\star$.

**Remarque 2.30.** *Dans toute la suite, nous utiliserons les notations suivantes :*
- *$\varepsilon^\star$ désignera l'erreur $\|p^\star - g\|_\infty$ où $p^\star$ est un polynôme de meilleure approximation* à coefficients réels *;*
- *$\varepsilon_{opt}$ désignera l'erreur $\|p_{opt} - g\|_\infty$ où $p_{opt}$ est un polynôme optimal parmi les polynômes dont les coefficients sont représentables sur les formats demandés ;*
- *$\varepsilon_{cible}$ désignera une erreur qu'on cherche à atteindre : sans chercher nécessairement à trouver un polynôme optimal $p_{opt}$, on peut se contenter d'un polynôme $p$ tel que*

$$\|p - g\|_\infty \leq \varepsilon_{cible}.$$

Reprenons l'exemple 2.29 : il nous servira de fil rouge durant cette section. Le polynôme de meilleure approximation à coefficients réels fournit une erreur de $7,9\,\mathrm{e}{-25}$. Puisque cette erreur est optimale, nous savons en particulier qu'aucun polynôme à coefficients machine ne peut fournir une erreur meilleure que $7,9\,\mathrm{e}{-25}$. Si on désire une erreur plus petite, il faut impérativement augmenter le degré.

Soit $d \geq n$ un entier. Choisissons (arbitrairement, dans un premier temps) $d+1$ points distincts $x_0, \ldots, x_d$ dans l'intervalle $[a, b]$ et une erreur cible $\varepsilon_{\text{cible}} \geq \varepsilon^\star$. On considère alors le problème suivant, d'inconnues $a_0, \ldots, a_n$ :

$$\begin{cases} \left(\sum_{j=0}^{n} a_j\,\varphi_j(x_0)\right) - g(x_0) & \leq \varepsilon_{\text{cible}} \\ g(x_0) - \left(\sum_{j=0}^{n} a_j\,\varphi_j(x_0)\right) & \leq \varepsilon_{\text{cible}} \\ \vdots & \\ \left(\sum_{j=0}^{n} a_j\,\varphi_j(x_d)\right) - g(x_d) & \leq \varepsilon_{\text{cible}} \\ g(x_d) - \left(\sum_{j=0}^{n} a_j\,\varphi_j(x_d)\right) & \leq \varepsilon_{\text{cible}}. \end{cases} \tag{2.6}$$

Nous avons construit le problème (2.6) en ne gardant qu'un nombre fini de contraintes linéaires (aux points $x_i$) sur l'infinité que constituent les points de l'intervalle $[a, b]$. Ce problème admet au moins une solution réelle : les coefficients $a_0^\star, \ldots, a_n^\star$ de $p^\star$.

Ce qui est important, c'est que, si $p$ est un « polynôme » tel que $\|p - g\|_\infty \leq \varepsilon_{\text{cible}}$, alors ses coefficients vérifient (2.6). Donc si une propriété est vérifiée par toutes les solutions de (2.6), elle est

forcément vérifiée par $p$. Ceci va nous permettre de déduire facilement des conditions nécessaires sur les coefficients de $p$.

En règle générale, l'ensemble des solutions de (2.6) est un polytope $\mathcal{P}$ de $\mathbb{R}^{n+1}$ (c'est le cas dès que le système formé par les $(\varphi_j(x_i))$ est de rang $n+1$, ce qui correspond au cas générique). La projection orthogonale du polytope sur chacun des axes constitue un outil puissant pour analyser la situation. Étant donné un indice $j$, il est possible de projeter le polytope $\mathcal{P}$ orthogonalement sur l'axe correspondant à $a_j$ : il suffit pour cela de résoudre le problème de programmation linéaire *minimiser ou maximiser $a_j$ sous la contrainte du système* (2.6). Pour résoudre ce problème de programmation linéaire, on peut par exemple utiliser l'algorithme du simplexe [Dan51, Sch86]. Si $u_j$ désigne le minimum et $v_j$ le maximum, on a par définition

$$\forall (a_0, \ldots, a_n) \in \mathbb{R}^{n+1} \text{ vérifiant (2.6)}, \; a_j \in [u_j, v_j]\,.$$

Serge Torres a programmé un outil qui utilise GLPK [5] pour calculer les projections orthogonales. À l'heure actuelle, cet outil est à l'état de prototype non distribué. Il prend en entrée la fonction à approcher, une liste de points $x_i$, la liste des monômes qu'on souhaite utiliser et l'erreur cible $\varepsilon_{\text{cible}}$. En retour, il calcule les intervalles $[u_j, v_j]$ $(j = 0, \ldots, n)$.

**Exemple 2.31.** *Reprenons notre exemple fil rouge (exemple 2.29). Ce que nous savons pour l'instant, c'est que $p^\star$ fournit une erreur de $7,9\,e{-}25$ et que le polynôme $\widehat{p}$ obtenu en arrondissant les coefficients de $p^\star$ au plus près fournit une erreur de $4\,035\,e{-}25$. Il est donc beaucoup moins bon. On aimerait savoir s'il existe un polynôme $p$ fournissant une erreur inférieure à $\varepsilon_{cible} = 10\,e{-}25$ par exemple.*
*On utilise l'outil de Torres, avec cette borne $\varepsilon_{cible}$. Pour les points $x_i$, on choisit 30 points de Tchebychev dans l'intervalle (voir l'équation (2.5) page 99). On obtient alors les bornes ci-dessous sur les coefficients.*
*La notation $a = 1234[5|6]$ est une notation abrégée pour signifier que $a$ appartient à l'intervalle $[12345, 12346]$. Il s'agit du même principe que le mode d'affichage `midpointmode` de Sollya. Cela permet de voir d'un coup d'œil les chiffres contraints dans le développement décimal de $a$.*

$$\begin{aligned}
a_0 &= 0,693147180559945309417 23[1|4]\\
a_1 &= 0,240226506959100712333[4|7]\\
a_2 &= 0,555041086648215799[3|5]\; e{-}1\\
a_3 &= 0,961812910762847[6|8]\; e{-}2\\
a_4 &= 0,13333558146428[5|7]\; e{-}2\\
a_5 &= 0,15403530393[3|4]\; e{-}3\\
a_6 &= 0,15252733[7|8]\; e{-}4\\
a_7 &= 0,1321548[6|8]\; e{-}5\\
a_8 &= 0,10178[4|6]\; e{-}6\\
a_9 &= 0,70[4|7]\; e{-}8
\end{aligned}$$

*On remarque alors que l'intervalle correspondant à $a_1$ ne contient aucun nombre représentable sur 64 bits. Par conséquent,* ***il n'existe aucun polynôme*** *dont le coefficient $a_1$ soit un* `double` *étendu et qui fournisse une erreur de $10\,e{-}25$.*

Au départ, nous savions uniquement que $\varepsilon_{\text{opt}}$ était minoré par $\varepsilon^\star$. En utilisant uniquement la projection orthogonale sur l'axe correspondant à $a_1$, nous venons de trouver le minorant $10\,e{-}25$, qui

5. outils pour la programmation linéaire distribués sous licence GPL : http://www.gnu.org/software/glpk/

est meilleur. On peut alors essayer de nouvelles valeurs de $\varepsilon_{\text{cible}}$ et tenter d'améliorer le minorant : par exemple, en prenant $\varepsilon_{\text{cible}} = 400\,\text{e}{-25}$, l'intervalle correspondant à $a_1$ ne contient toujours aucun `double` étendu, et donc $\varepsilon_{\text{opt}} \geq 400\,\text{e}{-25}$.

Ainsi, le minorant initial donné par $\varepsilon^\star \simeq 7,9\,\text{e}{-25}$ était très en dessous de la réalité. Imaginons que nous trouvions un polynôme fournissant une erreur de $445\,\text{e}{-25}$ (nous décrirons en section 2.3.4 et suivantes un algorithme capable de nous donner rapidement un tel polynôme) : on saura alors que $400\,\text{e}{-25} \leq \varepsilon_{\text{opt}} \leq 445\,\text{e}{-25}$. On peut éventuellement décider de s'en contenter sans nécessairement chercher à trouver un polynôme réellement optimal. Nous retrouvons ici le même raisonnement que celui que nous avions utilisé avec le théorème de La Vallée Poussin (voir l'équation (2.2) page 84).

Dans le processus de développement de code pour évaluer une fonction numérique, on a rarement une liste de formats imposés d'avance ; on a généralement besoin de fixer les formats soi-même, en fonction des contraintes imposées par l'architecture cible. En règle générale, on essaie de privilégier les formats ayant la plus faible précision. En particulier, on souhaite n'utiliser des expansions que lorsque c'est réellement indispensable. Souvent, cette information peut être obtenue en utilisant la projection orthogonale correspondante. S'il est vraiment nécessaire d'utiliser une expansion pour un certain coefficient $a_j$, l'intervalle projeté $[u_j, v_j]$ est généralement tellement fin qu'il ne contient aucun nombre au format `double`. Dans ce cas, il est *nécessaire* d'utiliser au moins un double-`double`.

**Exemple 2.32.** *Continuons notre exemple fil rouge. Pour l'instant, nous savons que* $\varepsilon_{opt}$ *est compris entre* $400\,e{-25}$ *et* $4\,035\,e{-25}$ *(car c'est l'erreur fournie par le polynôme* $\widehat{p}$*). Dans l'énoncé de l'exemple, il est demandé que le coefficient* $a_0$ *soit un double-`double`. Peut-être un `double` aurait-il suffi ? En utilisant l'outil de Torres, on construit le polytope correspondant à* $\varepsilon_{cible} = 4\,035\,e{-25}$ *(toujours en utilisant* 30 *points de Tchebychev). En projetant par rapport à* $a_0$ *on obtient* $a_0 = 0,6931471805599453094\,1[6|8]$.
*Il est facile de voir que cet intervalle ne contient aucun `double` et même aucun `double` étendu. Pour atteindre l'erreur* $4\,035\,e{-25}$*, il faut donc au moins utiliser un double-`double`. Mieux : la procédure pour arrondir* $a$ *au double-`double`* $a_h + a_\ell$ *le plus proche consiste à effectuer la séquence :*

$$a_h = \circ(a);$$
$$a_\ell = \circ(a - a_h);$$

*où* $\circ(\cdot)$ *est l'arrondi au `double` le plus proche. Or tous les points de l'intervalle* $0,6931471805599453094\,1[6|8]$ *s'arrondissent au même `double` :*

$$a_{0,h} = 6243314768165359 \cdot 2^{-53}.$$

*Par conséquent,* $a_{0,h}$ *n'est plus une inconnue et nous n'avons qu'à chercher la valeur de* $a_{0,\ell}$*. Notre problème devient donc le suivant : approcher la fonction* $g - a_{0,h}$ *en erreur absolue sur* $[-1/16, 1/16]$ *par un polynôme de degré* 9*. Le coefficient constant doit être un `double` et les autres des `double` étendus.*

La plupart du temps, les expansions de nombres flottants peuvent se traiter de cette manière. Le principe s'étend aux triples-`double` $(a_h + a_m + a_\ell)$ pour lesquels on peut généralement déterminer les valeurs de $a_h$ et $a_m$.

Dans le même ordre d'idée, si un certain coefficient $a_j$ doit être représenté par un nombre flottant sur $t_j$ bits, il arrive parfois que la projection $[u_j, v_j]$ ne contienne qu'un seul nombre sur $t_j$ bits. Dans ce cas, la valeur de $a_j$ est donc fixée et n'est plus une inconnue.

**Exemple 2.33.** *Dans notre exemple fil rouge, supposons que nous ayons trouvé un polynôme $p_0$ dont l'erreur soit $532\,e{-}25$ (nous verrons comment l'obtenir facilement : voir page 129). En utilisant l'outil de Torres avec cette borne (et toujours 30 points de Tchebychev) on obtient l'encadrement $a_1 = 0,2402265069591007123[2|5]$.*
*Cet intervalle ne contient que deux* `double` *étendus :*

$$a_{1,a} = 17725587574382949699 \cdot 2^{-66} \quad et \quad a_{1,b} = 17725587574382949700 \cdot 2^{-66}.$$

*On peut alors calculer le polynôme à coefficients réels $p_a^\star$ de meilleure approximation de la fonction $g$ parmi ceux dont le coefficient $a_1$ est fixé à $a_{1,a}$ et de même le polynôme $p_b^\star$ correspondant à la valeur $a_{1,b}$.*
*Les erreurs correspondantes sont $\|p_a^\star - g\|_\infty \simeq 444\,e{-}25$ et $\|p_b^\star - g\|_\infty \simeq 497\,e{-}25$. Autrement dit : si l'on souhaite atteindre une erreur inférieure à $497\,e{-}25$, le coefficient $a_1$ ne peut prendre qu'une seule valeur : $a_1 = a_{1,a}$.*
*Au passage, observons que nous avons amélioré notre minorant de $\varepsilon_{opt}$ : nous savons désormais que $\varepsilon_{opt} \geq 444\,e{-}25$.*

Jusqu'à présent, nous avons toujours formulé le problème sous la forme *Étant donné une liste de formats, trouver le polynôme d'erreur minimale dont les coefficients respectent les formats.* En pratique, la situation est souvent inverse : on cherche à descendre en dessous d'une certaine erreur et on choisit des formats appropriés pour chaque coefficient. On peut utiliser les projections orthogonales pour avoir une idée de la précision nécessaire sur chaque coefficient.

**Exemple 2.34.** *Imaginons que, dans notre exemple fil rouge, on veuille simplement atteindre une erreur inférieure à $\varepsilon_{cible} = 532\,e{-}25$ avec le choix libre sur les précisions de chaque coefficient. On calcule, pour chaque $a_j$, la projection correspondante $[u_j, v_j]$ puis on cherche la plus petite précision $t_j$ pour laquelle il existe au moins un nombre en précision $t_j$ dans $[u_j, v_j]$. On sait alors que, pour atteindre l'erreur $\varepsilon_{cible}$, il est nécessaire que la précision du format de $a_j$ soit supérieure ou égale à $t_j$. Dans notre cas, on obtient :*

$$\begin{aligned} t_0 &= 71,\\ t_1 &= 62,\\ t_2 &= 54,\\ t_3 &= 46,\\ t_4 &= 40,\\ t_5 &= 30,\\ t_6 &= 23,\\ t_7 &= 11,\\ t_8 &= 11,\\ t_9 &= 0. \end{aligned}$$

*On peut donc essayer de chercher un polynôme d'approximation tel que $a_0$ soit un double-*`double`*, $a_1$ et $a_2$ des* `double` *étendus, $a_3$ à $a_6$ des* `double`*, et $a_7$, $a_8$ et $a_9$ des nombres au format* simple précision *(24 bits). Pour $a_6$, on pourrait peut-être choisir un nombre* simple précision*, mais, comme il y a peu de marge, ce n'est pas sûr (il faudrait appliquer la même démarche que nous avons eue pour le coefficient $a_1$ pour voir si $a_6$ a une valeur contrainte fixée ou non).*

*Effectivement, on peut trouver un polynôme satisfaisant à ces contraintes de format et avec une erreur de* $480\,e{-}25$ *(ce polynôme est obtenu avec l'algorithme que nous décrirons en section 2.3.4 et suivantes).*

### 2.3.3 Résolution exacte du problème.

Nous avons vu que les projections orthogonales donnaient un outil puissant pour déduire des contraintes sur les polynômes recherchés et pour trouver des minorants et majorants de $\varepsilon_{\text{opt}}$. Cette technique nous permet donc de déterminer une bonne estimation de la valeur de $\varepsilon_{\text{opt}}$.

À présent, nous nous posons la question de la résolution exacte du problème : nous voulons déterminer la valeur exacte de $\varepsilon_{\text{opt}}$ et la liste de tous les polynômes dont les coefficients vérifient les contraintes de formats et dont l'erreur est $\varepsilon_{\text{opt}}$.

Comme nous l'avons vu, les projections orthogonales permettent en général de déterminer les parties hautes des expansions de flottants. On est donc ramené à la recherche de polynômes dont les coefficients sont représentables sur un format de type *virgule fixe* ou de type *virgule flottante*. Si un coefficient $a$ est en virgule flottante sur $t$ bits, il s'écrit

$$a = \frac{m}{2^t}\,2^e \quad \text{(voir définition en page 34)}.$$

Il y a donc *a priori* deux inconnues : la mantisse $m$ et l'exposant $e$. Le cas de l'exposant n'est pas un gros problème : en effet, l'exposant d'un nombre représente simplement son ordre de grandeur, information que nous pouvons avoir via les projections orthogonales.

**Exemple 2.35.** *Nous laissons provisoirement de côté notre exemple fil rouge. Nous allons étudier l'exemple 2.26. Au départ, on sait simplement que* $8,3\,e{-}10 \leq \varepsilon_{opt} \leq 119\,e{-}10$ *: le minorant est donné par l'erreur du polynôme* $p^\star$ *de meilleure approximation à coefficients réels, et le majorant par l'erreur du polynôme arrondi* $\widehat{p}$*.*
*On peut obtenir un bien meilleur polynôme que* $\widehat{p}$ *(et donc un bien meilleur majorant) en utilisant l'algorithme que nous décrirons en section 2.3.4 et suivantes. En l'occurrence, on obtient le polynôme*

$$p_0 = 1 - \frac{8388607}{2^{24}}\,x + \frac{11628935}{2^{27}}\,x^2 + \frac{7234987}{2^{40}}\,x^3 - \frac{470491}{2^{28}}\,x^4 + \frac{6700979}{2^{38}}\,x^5 + \frac{2878101}{2^{36}}\,x^6.$$

*Il fournit une erreur* $\varepsilon_0 \simeq 10,33\,e{-}10$ *qui nous donne donc un majorant de* $\varepsilon_{opt}$ *bien plus fin. En utilisant l'outil de Torres avec* $20$ *points de Tchebychev et* $\varepsilon_{cible} = \varepsilon_0$*, on obtient les encadrements suivants :*

$$\begin{aligned}
a_0 &\in [0,999999998966,\ 1,0000000009845916]\,,\\
a_1 &= -0,4999999[9|0],\\
a_2 &= 0,8664[1|3]\ e{-}1,\\
a_3 &\in [4,822952501198682\ e{-}6,\ 8,352048514013137\ e{-}6]\,,\\
a_4 &= -0,17[6|4]\ e{-}2,\\
a_5 &= 0,2[1|8]\ e{-}4,\\
a_6 &= 0,4[0|3]\ e{-}4.
\end{aligned}$$

*L'exposant d'un nombre flottant* $a$ *est donné par* $e = \lfloor \log_2(a) \rfloor + 1$*. Dans l'intervalle encadrant* $a_1$ *par exemple, tous les flottants ont le même exposant* $e = -1$*. L'exposant de* $a_1$ *est donc fixé et n'est plus une inconnue.*

*De même, l'exposant de $a_2$ est $-3$, celui de $a_4$ est $-9$, celui de $a_5$ est $-15$ et celui de $a_6$ est $-14$.*
*Pour $a_0$ et $a_3$, la situation est plus délicate : les intervalles correspondants contiennent une puissance de 2 et il y a donc* a priori *deux exposants possibles pour chacun.*
*En ce qui concerne $a_0$, l'unique nombre en simple précision dans l'intervalle est 1. La valeur de $a_0$ est donc fixée à 1. On peut calculer le polynôme $p_0^\star$ de meilleure approximation à coefficients réels et tel que $a_0 = 1$ : l'erreur obtenue est $9,05\,e{-}10$ et constitue une nouvelle minoration de $\varepsilon_{opt}$.*
*Pour $a_3$, on ne peut pas conclure sur la base de l'encadrement calculé. Cependant, lors de la construction du polytope, nous n'avons pas porté de soin spécifique au choix des points $x_i$ ; nous avons juste utilisé 20 points de Tchebychev. Or le polytope que nous avons construit représente l'ensemble des polynômes $p$ pour lesquels $\forall i,\ |p(x_i) - g(x_i)| \leq \varepsilon_{cible}$. Nous avons donc intérêt à inclure, parmi les points $x_i$, des points où $|p - g|$ risque de dépasser $\varepsilon_{cible}$, c'est-à-dire les extrema de $|p - g|$.*
*On peut penser que le polynôme $p$ que l'on recherche est assez proche de $p_0^\star$ et il semble donc judicieux d'avoir, parmi les $x_i$, les extrema de $p_0^\star - g$. Un calcul numérique approché donne la liste suivante :*

$$\left(0,\ \frac{13034}{338147},\ \frac{13305}{73036},\ \frac{549055}{1417707},\ \frac{60127}{98217},\ \frac{2015303}{2479273},\ \frac{71685}{75388},\ 1\right).$$

*En utilisant l'outil de Torres sur cette liste de points (et toujours avec $\varepsilon_{cible} = \varepsilon_0$), on obtient (entre autres) les encadrements*

$$a_1 = -0.4999999[7|3] \quad \text{et} \quad a_3 \in [5,36278728560402\,e{-}6,\ 6,9321401399335\,e{-}6]\,.$$

*Cette fois, tous les flottants de l'encadrement de $a_3$ ont pour exposant $-17$. En outre, il n'y a plus qu'un seul nombre en simple précision possible pour $a_1$ : $a_1 = -8388607 \cdot 2^{-24}$. On peut donc calculer le polynôme $p_{01}^\star$ de meilleure approximation à coefficients réels et tel que $a_0 = 1$ et $a_1 = -8388607 \cdot 2^{-24}$ : l'erreur obtenue $10,02\,e{-}10$ constitue une nouvelle minoration de $\varepsilon_{opt}$.*

Comme on a pu le voir sur cet exemple, les projections orthogonales permettent en général de déterminer l'exposant de chaque coefficient. Il arrive parfois que l'encadrement donné par les projections soit trop large (comme c'était le cas pour $a_3$ dans notre exemple), mais ce cas est rare et on peut espérer n'avoir pas plus que deux exposants, qu'on pourra donc traiter séparément.

Deux cas particuliers méritent d'être mentionnés.

**Premier cas :** l'encadrement $[u_j, v_j]$ d'un certain coefficient $a_j$ peut contenir une puissance de 2 (c'était le cas de $a_0$ dans l'exemple). Il se peut (comme dans l'exemple) que ce soit alors la seule valeur possible pour $a_j$. Mais il se peut aussi que l'intervalle $[u_j, v_j]$, bien que petit, contienne quelques autres valeurs possibles. Cela signifie que $a_j$ sera finalement très proche d'une puissance de 2. Dans ce cas, il peut être intéressant en pratique de forcer la valeur $a_j$ à être cette puissance. Le polynôme obtenu sera plus simple à stocker et, lors de son évaluation, la multiplication par $a_j$ pourra être remplacée par un décalage.

**Second cas :** de même, l'encadrement $[u_j, v_j]$ contient parfois 0. Ce cas est plus ennuyeux en pratique parce qu'une infinité d'exposants est alors possible. Là encore, si l'encadrement $[u_j, v_j]$ est assez fin, il peut être intéressant de forcer la valeur de $a_j$ à 0. On résout ainsi le problème de la détermination de l'exposant de $a_j$ tout en simplifiant l'expression et l'évaluation du polynôme.

Nous verrons aussi en section 2.3.7 une méthode heuristique pour déterminer les exposants.

Cette méthode est bien moins lourde à mettre en œuvre que la technique des projections orthogonales. Cependant, en l'utilisant, on risque de se tromper et de ne pas trouver les exposants optimaux puisqu'elle n'est qu'heuristique.

En déterminant tous les exposants, nous nous sommes ramenés au cas où tous les formats sont la virgule fixe. Nous reformulons donc notre problème initial sous la forme suivante : on se donne pour chaque coefficient $a_j$ un entier relatif $E_j$ et on cherche $a_j$ sous la forme $m_j/2^{E_j}$ où $m_j$ est un entier quelconque. Quitte à remplacer la fonction de base $\varphi_j$ par $\varphi_j/2^{E_j}$, nous pouvons oublier les contraintes de format et chercher uniquement des « polynômes » à coefficients entiers. Nous supposerons donc désormais que les inconnues $a_0, \ldots, a_n$ sont des entiers quelconques.

### Approches précédentes

Le problème, tel que nous venons de le formuler, a été étudié dans le cas très particulier où $\varphi_j(x) = x^j$ et $g = 0$ ; autrement dit, on cherche un polynôme (dans la base usuelle des monômes $1, x, \ldots, x^n$) à coefficients entiers dont la norme sup soit la plus petite possible. Évidemment, le polynôme identiquement nul a une norme nulle : on demande donc de trouver un polynôme non trivial. Ces polynômes entiers de norme minimale sont appelés *polynômes entiers de Tchebychev*. Ils ont une importance en approximation diophantienne [Wu02]. Leurs propriétés théoriques ont été bien étudiées et des algorithmes existent pour les calculer pour des valeurs de $n$ modérées [BE96, HS97]. Les techniques utilisées pour les polynômes entiers de Tchebychev sont apparentées à celles que nous allons présenter : elles sont fondées sur la programmation linéaire et l'algorithme LLL.

Le problème général de l'approximation d'une fonction par un polynôme dont les coefficients sont en virgule fixe a été semble-t-il peu étudié. Kodek s'est intéressé à un problème similaire en théorie du signal [Kod80] : la synthèse de filtre à réponse impulsionnelle finie (FIR) avec des coefficients en virgule fixe. Il s'est aussi intéressé [Kod02] au problème dans sa généralité pour n'importe quel système remplissant la condition de Haar. Il utilise principalement des techniques de programmation linéaire et semble être limité à des précisions assez faibles (de l'ordre d'une dizaine de bits par coefficient).

Brisebarre, Muller et Tisserand ont proposé [BMT06] un algorithme utilisant la programmation linéaire en nombres rationnels pour l'approximation polynomiale à coefficients en virgule fixe. L'algorithme que nous allons présenter à présent est une extension de ce travail.

### Approche dichotomique

Revenons à présent au polytope $\mathcal{P}$ avec lequel nous travaillons depuis le début : un point $(a_0, \ldots, a_n)$ à coordonnées entières dans $\mathcal{P}$ correspond à un polynôme $p$ tel que

$$\forall i \in [\![0, d]\!], \ |p(x_i) - g(x_i)| \leq \varepsilon_{\text{cible}}.$$

Bien évidemment, cela ne signifie pas forcément que $\|p - g\|_\infty \leq \varepsilon_{\text{cible}}$. Il faut donc faire le tri, parmi les points à coordonnées entières dans le polytope, entre ceux pour lesquels on a effectivement $\|p - g\|_\infty \leq \varepsilon_{\text{cible}}$ et les autres. Il se peut même, si on a choisi $\varepsilon_{\text{cible}} < \varepsilon_{\text{opt}}$, que le polytope contienne beaucoup de points à coordonnées entières, bien qu'aucun d'entre eux ne soit solution du problème qui nous intéresse.

**Proposition 2.36.** *On se donne une borne $\varepsilon_{cible}$ et un point $(a_0, \ldots, a_n)$ à coordonnées entières dans le polytope $\mathcal{P}$ associé. Soit $p = \sum_{j=0}^{n} a_j \varphi_j$. Pour que le « polynôme » $p$ soit optimal, il suffit que*

$$\begin{cases} \|p - g\|_\infty \leq \varepsilon_{cible} \\ \forall (b_0, \ldots, b_n) \in \mathbb{Z}^{n+1} \cap \mathcal{P}, \ \|p - g\|_\infty \leq \|q - g\|_\infty \quad \textit{en notant } q = \displaystyle\sum_{j=0}^{n} b_j \varphi_j. \end{cases}$$

*Démonstration.* Supposons que $p$ vérifie l'hypothèse. Puisque $\|p-g\|_\infty \leq \varepsilon_{\text{cible}}$, on a l'inégalité $\varepsilon_{\text{opt}} \leq \|p-g\|_\infty \leq \varepsilon_{\text{cible}}$. Il suit que pour tout « polynôme » optimal $p_{\text{opt}}$ et tout indice $i$, on a

$$|p_{\text{opt}}(x_i) - g(x_i)| \leq \varepsilon_{\text{opt}} \leq \varepsilon_{\text{cible}}.$$

Donc tous les « polynômes » optimaux sont dans le polytope $\mathcal{P}$. Comme $\|p-g\|_\infty$ est minimal parmi les « polynômes » à coefficients entiers contenus dans le polytope $\mathcal{P}$, il suit que $p$ est optimal. □

**Proposition 2.37.** *Si, pour tout point $(a_0, \ldots, a_n)$ à coordonnées entières dans le polytope $\mathcal{P}$, le « polynôme » $p$ associé vérifie $\|p-g\|_\infty > \varepsilon_{cible}$, alors $\varepsilon_{opt} > \varepsilon_{cible}$.*

*Démonstration.* Par contraposée : si $\varepsilon_{\text{opt}} \leq \varepsilon_{\text{cible}}$, notons $p_{\text{opt}}$ un polynôme à coefficients entiers optimal. Alors pour tout $i$, $|p_{\text{opt}}(x_i) - g(x_i)| \leq \varepsilon_{\text{opt}} \leq \varepsilon_{\text{cible}}$. Donc les coefficients de $p_{\text{opt}}$ constituent un point à coordonnées entières dans $\mathcal{P}$ pour lequel $\|p_{\text{opt}} - g\|_\infty \leq \varepsilon_{\text{cible}}$. □

De ces propositions découle un algorithme simple pour trouver $\varepsilon_{\text{opt}}$ et la liste des polynômes optimaux : étant donné $\varepsilon_{\text{cible}}$, on cherche tous les points à coordonnées entières du polytope $\mathcal{P}$ correspondant, et on calcule à chaque fois la norme sup de l'erreur associée. Soit $\varepsilon_{\min}$ la norme sup minimale :

- si $\varepsilon_{\min} \leq \varepsilon_{\text{cible}}$, on a trouvé l'erreur optimale d'après la proposition 2.36. La liste des points à coordonnées entières dans le polytope qui fournissent cette erreur est la liste des « polynômes » optimaux ;
- sinon, on a $\varepsilon_{\text{cible}} < \varepsilon_{\text{opt}} \leq \varepsilon_{\min}$ d'après la proposition 2.37. On peut donc recommencer en prenant une nouvelle valeur de $\varepsilon_{\text{cible}}$ intermédiaire entre les deux.

Torres a implémenté un deuxième outil qui permet de parcourir les points à coordonnées entières dans le polytope. Là encore, il s'agit uniquement d'un prototype qui n'est pas distribué pour l'instant. Il repose sur l'utilisation de l'outil PIP [6]. Ce logiciel permet d'obtenir une représentation paramétrique de tous les points à coordonnées entières dans un polytope. À partir de cette représentation, l'outil de Torres parcourt la liste des points pour trouver celui correspondant à la plus petite erreur.

**Exemple 2.38.** *Continuons de traiter l'exemple 2.26 : nous savons pour l'instant que le polynôme recherché est de la forme*

$$p(x) = 1 - \frac{8388607}{2^{24}}\,x + \frac{m_2}{2^{27}}\,x^2 + \frac{m_3}{2^{41}}\,x^3 + \frac{m_4}{2^{33}}\,x^4 + \frac{m_5}{2^{39}}\,x^5 + \frac{m_6}{2^{38}}\,x^6,$$

*où les $m_j$ sont des entiers. En outre, nous savons que $10,02\,e{-}10 \leq \varepsilon_{opt} \leq 10,33\,e{-}10$.*
***Première étape :** nous commençons en posant $\varepsilon_{cible} = 10,17\,e{-}10$ (c'est-à-dire approximativement le milieu entre $10,02\,e{-}10$ et $10,33\,e{-}10$) et en utilisant la liste de points de l'exemple précédent :*

$$\left(0, \frac{13034}{338147}, \frac{13305}{73036}, \frac{549055}{1417707}, \frac{60127}{98217}, \frac{2015303}{2479273}, \frac{71685}{75388}, 1\right).$$

*On commence à parcourir les points à coordonnées entières du polytope $\mathcal{P}$. Comme il semble y en avoir un trop grand nombre (plusieurs dizaines de milliers), on décide d'arrêter le parcours et on en choisit un quelconque (au hasard) :*

$$p_1(x) = 1 - \frac{8388607}{2^{24}}\,x + \frac{11628931}{2^{27}}\,x^2 + \frac{14970912}{2^{41}}\,x^3 - \frac{15060833}{2^{33}}\,x^4 + \frac{13757155}{2^{39}}\,x^5 + \frac{11444233}{2^{38}}\,x^6.$$

6. distribué sous licence GPL et disponible à l'adresse http://www.piplib.org/

*Le graphe de l'erreur correspondante est représenté en figure 2.8.*
*L'erreur de $p_1$ est $\|p_1 - g\|_\infty \simeq 10,53\, e{-}10$ : il faut ajouter des points dans la construction du polytope, de façon à empêcher l'erreur de dépasser le seuil de $10,17\, e{-}10$. Le plus logique est d'ajouter les points $x_i$ correspondant à ceux des extrema locaux de $p_1 - g$ dont la valeur absolue est plus grande que $10,17\, e{-}10$. On calcule donc des valeurs approchées de ces extrema et on les ajoute à la liste.*

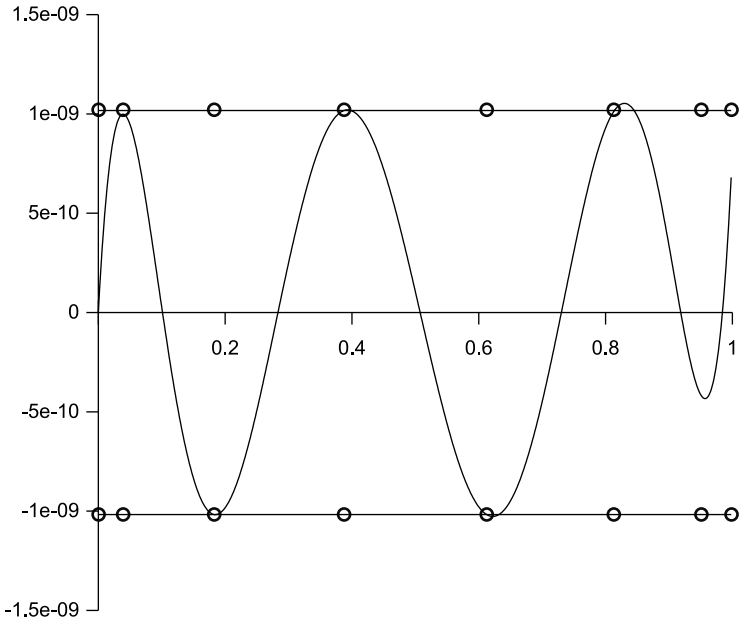



**Figure 2.8 :** Le graphe de l'erreur du polynôme $p_1$ ainsi que les points utilisés pour construire le polytope $\mathcal{P}$. L'erreur dépasse la borne $10,17\,\mathrm{e}{-}10$ qui était fixée : il faut ajouter des points.

***Deuxième étape :*** *on recommence donc avec la nouvelle liste de points :*

$$\left(0,\, \frac{13034}{338147},\, \frac{13305}{73036},\, \frac{19826}{107521},\, \frac{549055}{1417707},\, \frac{25765}{65573},\, \frac{60127}{98217},\, \frac{21652}{34807},\, \frac{2015303}{2479273},\, \frac{20831}{25108},\, \frac{71685}{75388},\, 1\right).$$

*Le polytope contient encore des milliers de points à coordonnées entières. On choisit l'un d'entre eux (à nouveau au hasard) :*

$$p_2(x) = 1 - \frac{8388607}{2^{24}}\, x + \frac{11628931}{2^{27}}\, x^2 + \frac{14972697}{2^{41}}\, x^3 - \frac{15060883}{2^{33}}\, x^4 + \frac{13763947}{2^{39}}\, x^5 + \frac{11442040}{2^{38}}\, x^6.$$

*Cette fois l'erreur de $p_2$ est environ $10,1699\, e{-}10$ : on sait donc à présent que*

$$10,02\, e{-}10 \leq \varepsilon_{opt} \leq 10,1699\, e{-}10.$$

***Troisième étape :*** *on choisit à présent une nouvelle valeur de $\varepsilon_{cible}$ intermédiaire entre les deux : $\varepsilon_{cible} = 10,10\, e{-}10$. On garde la même liste de points. À nouveau, il y a des milliers de points à coordonnées entières. On en choisit un :*

$$p_3(x) = 1 - \frac{8388607}{2^{24}}\, x + \frac{11628932}{2^{27}}\, x^2 + \frac{14809046}{2^{41}}\, x^3 - \frac{15058779}{2^{33}}\, x^4 + \frac{13587023}{2^{39}}\, x^5 + \frac{11481810}{2^{38}}\, x^6.$$

*L'erreur de $p_3$ est environ $10,15\, e{-}10$. Comme à la fin de la première étape, on ajoute des points correspondant approximativement aux extrema de $p_3 - g$ dont la valeur absolue est plus grande que $10,10\, e{-}10$.*

***Quatrième étape :*** *on ajoute donc les points suivants :*

$$\frac{33503}{180435},\ \frac{18229}{46033},\ \textit{et}\ \frac{19208}{20311}.$$

*On recommence avec la même erreur cible. À nouveau, il y a des milliers de points. On choisit l'un d'eux :*

$$p_4(x) = 1 - \frac{8388607}{2^{24}}\,x + \frac{11628932}{2^{27}}\,x^2 + \frac{14809440}{2^{41}}\,x^3 - \frac{15058788}{2^{33}}\,x^4 + \frac{13587967}{2^{39}}\,x^5 + \frac{11481577}{2^{38}}\,x^6.$$

*Son erreur est environ* $10,0999\,e{-}10$. *On sait donc désormais que*

$$10,02\,e{-}10 \leq \varepsilon_{opt} \leq 10,0999\,e{-}10.$$

***Cinquième étape :*** *on essaie donc à présent avec* $\varepsilon_{cible} = 10,06\,e{-}10$ *et la même liste de points. Cette fois, le polytope* $\mathcal{P}$ *est vide. On peut donc affirmer que*

$$10,06\,e{-}10 \leq \varepsilon_{opt} \leq 10,0999\,e{-}10.$$

***Sixième étape :*** *on essaie avec* $\varepsilon_{cible} = 10,08\,e{-}10$ *et la même liste de points. Il y a des milliers de points à coordonnées entières. En voici un :*

$$p_5(x) = 1 - \frac{8388607}{2^{24}}\,x + \frac{11628932}{2^{27}}\,x^2 + \frac{14811621}{2^{41}}\,x^3 - \frac{15058830}{2^{33}}\,x^4 + \frac{13591943}{2^{39}}\,x^5 + \frac{11480659}{2^{38}}\,x^6.$$

*Son erreur est environ* $10,08002\,e{-}10$. *On sait donc que*

$$10,06\,e{-}10 \leq \varepsilon_{opt} \leq 10,08002\,e{-}10.$$

*Comme cette erreur n'est que très légèrement supérieure à* $10,08\,e{-}10$, *on peut essayer de baisser* $\varepsilon_{cible}$ *tout en ajoutant les points où l'erreur de* $p_5$ *dépasse* $10,08\,e{-}10$.
***Septième étape :*** *il n'y a qu'un seul point à ajouter :* $10670/26969$. *On pose à présent* $\varepsilon_{cible} = 10,07\,e{-}10$ *et ainsi de suite.*

En poursuivant ainsi l'exemple, on peut montrer que l'unique polynôme optimal est

$$p_{\text{opt}}(x) = 1 - \frac{8388607}{2^{24}}\,x + \frac{11628932}{2^{27}}\,x^2 + \frac{14813102}{2^{41}}\,x^3 - \frac{15058857}{2^{33}}\,x^4 + \frac{13594231}{2^{39}}\,x^5 + \frac{11480194}{2^{38}}\,x^6$$

dont l'erreur est environ $10,06572652\,\text{e}{-}10$.

Cette méthode a le mérite d'être exacte et de fournir la liste complète des polynômes optimaux. En revanche elle est un peu laborieuse et très coûteuse en temps de calcul : il faut choisir un point dans le polytope, calculer la norme de l'erreur correspondante, rajouter des contraintes pour la construction du polytope, etc. et recommencer de nombreuses fois. À la fin, lorsque le polytope contient un nombre raisonnable de points à coordonnées entières, il faut les parcourir tous et pour chacun d'eux calculer la norme de l'erreur.

### 2.3.4 Résolution approchée du problème

Dans les exemples 2.35 et 2.38, nous avons utilisé un polynôme $p_0$ dont l'erreur $(10,33\,\text{e}{-}10)$ était déjà proche de la valeur optimale $(10,06\,\text{e}{-}10)$. Si nous ne l'avions pas eu, nous aurions été contraints de travailler avec le polynôme $\widehat{p}$ (obtenu en arrondissant les coefficients d'un minimax $p^\star$) dont l'erreur était $119\,\text{e}{-}10$. Les projections (exemple 2.35) auraient été beaucoup moins fines ; nous n'aurions sans doute pas pu montrer que les valeurs de $a_0$ et $a_1$ étaient contraintes. En outre, le

processus dichotomique pour s'approcher de l'erreur optimale (exemple 2.38) aurait été beaucoup plus lent.

Il est donc intéressant de pouvoir calculer rapidement un polynôme $p$ à coefficients entiers et de bonne qualité. Cela fournit un majorant assez fin de $\varepsilon_{\text{opt}}$ et permet de tirer des projections orthogonales plus d'informations qu'on en aurait eues en travaillant uniquement avec le polynôme $\widehat{p}$.

Par ailleurs, d'un simple point de vue pratique, on n'a pas nécessairement besoin d'avoir un polynôme optimal. Un polynôme de bonne qualité peut parfaitement faire l'affaire. Tout comme avec l'algorithme de Rémez, on peut se fixer à l'avance un paramètre $\mathsf{qualite} > 0$. Si on a pu trouver $\varepsilon_{\text{inf}}$ tel que $\varepsilon_{\text{opt}} \geq \varepsilon_{\text{inf}}$ et qu'on a un polynôme $p_0$ qui vérifie $\|p_0 - g\|_\infty \leq (1 + \mathsf{qualite})\,\varepsilon_{\text{inf}}$, on peut s'estimer satisfait et ne pas chercher de meilleur polynôme.

Dans un cas comme dans l'autre, on aimerait avoir un processus rapide pour déterminer un polynôme de bonne qualité $p_0$ à coefficients entiers approchant $g$. Ce processus peut être complètement heuristique ; s'il échoue, il est toujours possible de partir de $\widehat{p}$ et d'utiliser la méthode exacte exposée dans la section précédente. En somme, nous voulons un algorithme :

— rapide ;
— donnant de bons résultats en pratique ;
— mais dont le comportement n'est pas nécessairement garanti.

Si l'algorithme échoue ou renvoie un polynôme approchant très mal la fonction $g$, on s'en apercevra en faisant un simple calcul de norme sup et on utilisera $\widehat{p}$.

Pour concevoir un tel algorithme, nous nous appuyons sur la remarque suivante : en règle générale, le polynôme que nous cherchons à obtenir se comporte de manière assez semblable au polynôme de meilleure approximation à coefficients réels $p^\star$. Considérons $n+1$ points $z_0, \ldots, z_n$ dans l'intervalle $[a, b]$. Nous les notons $z_i$ pour bien les différencier des $x_i$ de la section précédente qui n'ont rien à voir.

Considérons un polynôme $p$ interpolateur de $p^\star$ aux points $z_i$. Autrement dit, les coefficients $(a_0, \ldots, a_n)$ de $p$ vérifient le système linéaire suivant :

$$\begin{cases} \displaystyle\sum_{j=0}^{n} a_j \varphi_j(z_0) = p^\star(z_0) \\ \qquad\qquad \vdots \\ \displaystyle\sum_{j=0}^{n} a_j \varphi_j(z_n) = p^\star(z_n) \end{cases}$$

qu'on peut encore écrire

$$\sum_{j=0}^{n} a_j \begin{pmatrix} \varphi_j(z_0) \\ \vdots \\ \varphi_j(z_n) \end{pmatrix} = \begin{pmatrix} p^\star(z_0) \\ \vdots \\ p^\star(z_n) \end{pmatrix}. \tag{2.7}$$

Évidemment, $p^\star$ est la solution exacte de ce système. Mais on peut espérer plus généralement que, si $p$ est une solution approchée de ce système, les comportements de $p$ et $p^\star$ sur l'intervalle $[a, b]$ seront assez semblables.

Ainsi, ce qui nous intéresse, c'est de trouver une solution approchée (la meilleure possible) de ce système, sous la contrainte que les coefficients de la solution doivent être entiers. Notons $b_j$ le vecteur ${}^t(\varphi_j(z_0), \ldots, \varphi_j(z_n))$ et $v$ le vecteur ${}^t(p^\star(z_0), \ldots, p^\star(z_n))$. Nous cherchons donc une combinaison linéaire à coefficients entiers des vecteurs $b_j$ qui soit la plus proche possible du vecteur $v$.

Pour préciser ce que nous entendons par *proche*, il nous faut munir $\mathbb{R}^{n+1}$ d'une norme $\|\cdot\|$ ; nous mesurerons alors la proximité de $\sum a_j b_j$ et de $v$ par la norme $\|v - \sum a_j b_j\|$. En l'occurrence, la norme que nous mettons sur $\mathbb{R}^{n+1}$ n'a pas une grande importance puisque notre idée ne repose

sur rien d'autre qu'une intuition : une solution approchée du système (2.7) fournirait un polynôme $p$ qui ressemblerait à $p^\star$ au point que $\|p-g\|_\infty$ serait presque optimal. Nous utiliserons donc la norme euclidienne :

$$\left\| \begin{pmatrix} u_0 \\ \vdots \\ u_n \end{pmatrix} \right\| = \sqrt{u_0^2 + \cdots + u_n^2}.$$

L'ensemble $\Gamma = \{\sum_{j=0}^n a_j\, b_j,\ (a_0, \ldots, a_n) \in \mathbb{Z}^{n+1}\}$ est ce qu'on appelle *un réseau*. La structure algébrique de réseau a été largement étudiée et des algorithmes efficaces permettent de travailler avec. Comme nous allons le voir, l'algorithmique des réseaux repose sur des idées géométriques et il est donc naturel de travailler en norme euclidienne.

Notre problème revient à trouver, dans le réseau $\Gamma$, le point le plus proche du vecteur $v$. Ce problème est connu sous le nom de *problème du plus proche vecteur* ou encore CVP, de l'anglais *closest vector problem*. Comme nous allons le voir dans les sections suivantes, un algorithme appelé *algorithme de Babai* permet de résoudre de façon approchée le problème du plus proche vecteur. L'algorithme de Babai s'appuie lui-même sur l'algorithme LLL que nous allons présenter.

### 2.3.5 Généralités sur les réseaux euclidiens

Nous abordons à présent l'étude d'un outil central de notre méthode : les réseaux. Un réseau est une structure algébrique discrète, ce qui en fait un outil de modélisation très commode dans de nombreux problèmes informatiques. La théorie des réseaux trouve des applications notamment en cryptologie, mais aussi dans divers autres domaines de l'informatique (on pourra par exemple consulter [NS01], [Ste98a], [Odl91] ou [LSZ03]). Ils apparaissent naturellement dans de nombreuses circonstances, en particulier dans certains domaines de la physique, tels que la cristallographie. Notre méthode consiste à transposer le problème de la recherche d'un polynôme à coefficients flottants en la recherche d'un vecteur à coefficients entiers dans un certain espace vectoriel. La structure de réseau sera partout présente dans cette approche. Pour une présentation générale de la théorie des réseaux, on pourra lire [Cas97] ou [GL87].

**Définition 2.39** (Réseau)**.** On se donne un $\mathbb{R}$-espace vectoriel $E$ de dimension finie $d$ et une famille libre $(b_1, \cdots, b_p)$ de $E$. L'ensemble

$$\Gamma = \left\{ \sum_{j=1}^{p} \lambda_j b_j \right\}_{(\lambda_1, \cdots, \lambda_p) \in \mathbb{Z}^p}$$

est appelé réseau. La famille $(b_1, \cdots, b_p)$ est appelée une base de $\Gamma$.

Par exemple, $\mathbb{Z}$ est un réseau de l'espace $E = \mathbb{R}$, de base 1 ; $\mathbb{Z} \times \mathbb{Z}\sqrt{3}$ est un réseau de l'espace $E = \mathbb{R}^2$ de base

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \sqrt{3} \end{pmatrix}.$$

**Remarque 2.40.** *Lorsque $E = \mathbb{R}^d$, il est naturellement muni de sa base canonique. En écrivant les vecteurs dans cette base, on peut considérer la matrice $B \in \mathcal{M}_{d,p}(\mathbb{R})$ dont les colonnes sont les $b_j$ exprimés dans la base canonique. Notons alors que, si $\Lambda$ désigne le vecteur ${}^t(\lambda_1, \cdots, \lambda_p)$, $B\Lambda$ est l'expression du vecteur $\sum_{j=1}^p \lambda_j b_j$ dans la base canonique. En particulier, on a*

$$\Gamma = B \cdot \mathcal{M}_{p,1}(\mathbb{Z}).$$

> *En outre, si $M$ est une matrice à $p$ lignes et $r$ colonnes, la $k$-ième colonne du produit $BM$ est l'expression dans la base canonique du vecteur $\sum_{j=1}^{p} \lambda_j b_j$, où ${}^t(\lambda_1, \cdots, \lambda_p)$ désigne la $k$-ième colonne de $M$.*

Les réseaux partagent certaines propriétés classiques des espaces vectoriels :

**Proposition 2.41.** *Soient $(b_1, \cdots, b_p)$ et $(c_1, \cdots, c_q)$ deux bases d'un réseau $\Gamma$. Alors $p = q$ ; on appelle $p$ le rang du réseau. En outre, la matrice de passage de $B$ à $C$ est unimodulaire, c'est-à-dire qu'elle est dans $\mathcal{M}_{p,p}(\mathbb{Z})$ et de déterminant égal à $\pm 1$.*

*Démonstration.* Voir par exemple [Cas97]. □

En revanche, certains aspects des réseaux sont plus déroutants : un réseau $\Gamma$ de rang $p$ étant donné, il existe un sous-réseau strict $\Gamma' \subset \Gamma$ qui est aussi de rang $p$. Dit autrement : il existe des familles de $p$ éléments de $\Gamma$ qui sont libres mais qui ne constituent pas une base de $\Gamma$. Par exemple, dans le réseau $\mathbb{Z} \times \mathbb{Z}\sqrt{3}$, la famille

$$\begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \sqrt{3} \end{pmatrix}$$

est libre, mais elle engendre le réseau $2\,\mathbb{Z} \times \mathbb{Z}\sqrt{3}$.

Une conséquence notable de la proposition 2.41 est qu'un réseau de rang supérieur ou égal à 2 possède une infinité de bases : en effet, étant donné une base, on peut toujours en obtenir de nouvelles en multipliant la matrice associée par une matrice unimodulaire. Il faut avoir conscience que certaines bases sont plus adaptées que d'autres pour manipuler un réseau donné.

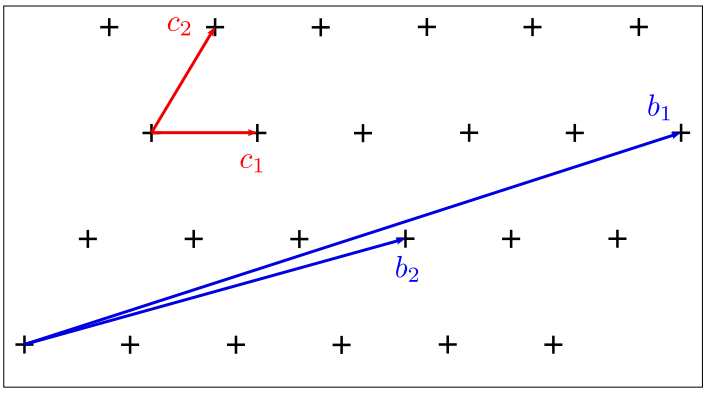



**Figure 2.9 :** Deux bases d'un même réseau

Par exemple, observons la figure 2.9 : les familles $(b_1, b_2)$ et $(c_1, c_2)$ sont deux bases d'un même réseau. Il apparaît clairement que la base $C$ est mieux adaptée car elle donne une vision locale du réseau : lorsqu'on ajoute un vecteur de base à un point du réseau, on obtient un point situé à proximité. En revanche, avec la base $B$, un vecteur de base nous éloigne considérablement du point de départ.

Si on cherche à explorer le voisinage d'un point dans un réseau (ce qui sera notre objectif dans peu de temps), une base est d'autant plus pratique à utiliser qu'elle est constituée de vecteurs courts : de cette façon, lorsqu'on ajoute un vecteur de base à un point du réseau, on ne s'écarte pas trop de ce point. Ainsi, la base $C$ sera mieux adaptée à l'exploration que la base $B$.

Cependant, lorsqu'un réseau nous est donné sous la forme d'une base, rien ne garantit *a priori* que cette base soit bien adaptée. Aussi faut-il apporter un soin particulier au choix de la base qu'on utilise pour manipuler le réseau. Mais comme on va le voir, il n'est pas aisé de définir précisément le concept de *base bien adaptée*.

### Plus courts vecteurs d'un réseau

Un premier problème algorithmique apparaît donc : comment trouver un vecteur non nul de norme minimale dans un réseau ? Ce problème est connu sous le nom de *shortest vector problem* (SVP). Il n'est pas évident *a priori* qu'un tel vecteur existe : il pourrait très bien y avoir une suite de vecteurs tous non nuls mais de normes tendant vers 0. Le théorème suivant justifie l'existence d'une solution à SVP :

**Théorème 2.42.** *Les réseaux de $\mathbb{R}^d$ sont exactement les sous-groupes additifs discrets de $\mathbb{R}^d$.*

**Corollaire 2.43.** *Dans tout réseau, il existe (au moins) un plus court vecteur non nul.*

*Démonstration.* Pour ces deux résultats, on consultera par exemple [Des86]. □

Il faut noter que le problème SVP dépend de la norme qu'on choisit sur $\mathbb{R}^d$. En dimension finie, toutes les normes d'un espace vectoriel sont équivalentes, c'est-à-dire (en particulier) qu'elles définissent la même topologie. Cela implique que *l'existence* d'un plus court vecteur est indépendante de la norme choisie. En revanche le plus court vecteur lui-même dépend de la norme. La figure 2.10 montre un exemple en dimension 2 avec la norme euclidienne, dont la boule est un disque, et avec la norme infini (la norme infini d'un vecteur est le maximum des valeurs absolues des composantes de ce vecteur), dont la boule est un carré.

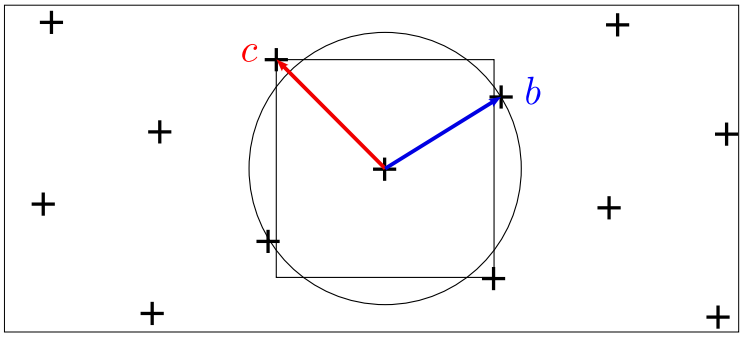



**Figure 2.10 :** Solutions différentes de SVP suivant la norme choisie : le vecteur $\vec{b}$ est le plus court vecteur en norme euclidienne mais pas en norme infini.

On notera que la solution à SVP n'est pas forcément unique [7] : par exemple, dans $\mathbb{Z}^2$ muni de la norme euclidienne habituelle, les vecteurs $(1, 0)$ et $(0, 1)$ sont deux solutions linéairement indépendantes de SVP.

Cependant, le théorème 2.42 et son corollaire ne sont pas constructifs. On a longtemps soupçonné — sans pouvoir précisément quantifier — que la recherche d'un plus court vecteur dans un réseau était un problème algorithmiquement dur. On sait depuis quelques années qu'il s'agit d'un problème NP-dur. [8]

Cela étant, ce qui nous intéressera par la suite est de posséder une base permettant de se déplacer localement dans le réseau, afin d'explorer le voisinage d'un point. Il n'est donc pas indispensable de connaître le plus court vecteur. D'ailleurs une base constituée d'un vecteur de norme minimale et de vecteurs beaucoup plus grands ne nous serait que de peu d'intérêt. Au contraire, on cherche à ce que *tous* les vecteurs de la base soient *assez* courts (il existe des problèmes pour lesquels seule la connaissance d'un vecteur très court est nécessaire ; mais dans notre cas, nous avons besoin d'une

7. Remarque : si $v$ est un plus court vecteur, $-v$ l'est aussi. Quand on parle d'unicité ici, c'est toujours modulo passage à l'opposé.

8. En réalité, le problème est NP-dur sous des réductions randomisées, et non pas NP-dur au sens classique. Pour plus de détails, voir par exemple [Ajt98], [Cai99] et [Mic01]).

base entièrement formée de vecteurs les plus courts possibles. Cette nécessité apparaîtra clairement lorsqu'on décrira l'algorithme de Babai qui calcule une approximation d'un plus proche vecteur dans un réseau). En outre, on souhaite que la base soit la plus orthogonale possible : ainsi, on pourra l'utiliser pour se déplacer avec la même aisance dans n'importe quelle direction de l'espace (en réalité l'orthogonalité et la longueur des vecteurs d'une base sont deux paramètres liés ; cependant on contrôlera préférentiellement l'un ou l'autre des deux paramètres suivant ce sur quoi on veut mettre l'accent).

Il n'y a pas de caractérisation universelle de ce qu'on pourrait appeler une *base idéale* d'un réseau. L'approche la plus naturelle consisterait à demander que le plus court vecteur soit dedans, ainsi que le plus court vecteur linéairement indépendant du premier, puis le plus court linéairement indépendant des deux premiers, et ainsi de suite. Pour la norme euclidienne, cette approche est valide jusqu'à la dimension 4. Mais à partir de la dimension 5, il se peut qu'aucune base ne vérifie cette propriété. Différentes notions de bases réduites ont alors été définies, qui collent plus ou moins à cet idéal inaccessible : citons par exemple la réduction de Hermite, de Minkowski, de Korkine-Zolotarev, etc. On pourra se reporter à [NS04] pour avoir un aperçu des différentes réductions et une liste de références plus complète.

Puisque SVP est NP-dur, aucune de ces réductions n'est effectivement calculable en un temps raisonnable. On cherche alors des critères de réduction approchée : on renonce à exiger d'avoir le plus court vecteur dans la base réduite, mais on essaie néanmoins d'obtenir, en un temps raisonnable, une base *assez* orthogonale et formée de vecteurs *assez* courts. On va voir que l'on peut atteindre cet objectif en temps polynomial avec l'algorithme LLL dont les résultats pratiques sont de très bonne qualité.

### 2.3.6 Algorithme LLL

En 1982, A. K. Lenstra, H. W. Lenstra Jr. et L. Lovász proposent une définition de réduction (la réduction LLL) et fournissent un algorithme (l'algorithme LLL) qui produit une base LLL-réduite en temps polynomial (voir [LLL82]).

Cet algorithme va être au centre de notre méthode car il donne une base bien adaptée pour se promener dans le réseau. Cette base n'est certes pas théoriquement parfaite, mais comme on le verra, elle est en pratique très souvent bien adaptée pour travailler dans le réseau. Pour une introduction à la réduction LLL, on pourra consulter [Lov86, Coh93].

L'objectif de l'algorithme LLL est d'obtenir une base la plus orthogonale possible ; la notion d'orthogonalité étant une notion purement euclidienne, c'est évidemment cette norme qui est choisie pour normer l'espace. Par la suite, sauf contre-indication explicite, la norme utilisée sera donc la norme euclidienne. Si $x = (x_1, \cdots, x_d)$ et $y = (y_1, \cdots, y_d)$ sont deux vecteurs, nous noterons $\langle x, y\rangle = \sum_i x_i y_i$ le produit scalaire de $x$ et $y$. Pour rappel, la norme euclidienne est définie par $\|x\|^2 = \langle x, x\rangle$.

À toute famille libre $(b_1, \cdots, b_p)$ d'un espace vectoriel euclidien est associée une famille orthogonale $(b_1^\star, \cdots, b_p^\star)$, appelée *famille de Gram-Schmidt*, qui vérifie $\forall\, i$, $\mathrm{Vect}(b_1, \cdots, b_i) = \mathrm{Vect}(b_1^\star, \cdots, b_i^\star)$. Cette famille est au cœur de l'algorithme LLL et il est nécessaire de se familiariser avec.

**Définition 2.44** (Famille de Gram-Schmidt)**.** On définit la famille de Gram-Schmidt associée à la famille $(b_1, \cdots, b_p)$ par récurrence de la façon suivante :

$$\begin{cases} b_1^\star = b_1 \\ b_{k+1}^\star = b_{k+1} - \displaystyle\sum_{j=1}^{k} \frac{\langle b_{k+1}, b_j^\star\rangle}{\langle b_j^\star, b_j^\star\rangle}\, b_j^\star \ . \end{cases}$$

Géométriquement, la somme dans la définition de $b_{k+1}^\star$ est la projection orthogonale du vecteur $b_{k+1}$ sur l'espace $E_k = \mathrm{Vect}(b_1^\star, \cdots, b_k^\star)$. Ainsi, $b_{k+1}^\star$ est la composante de $b_{k+1}$ orthogonale à $E_k$.

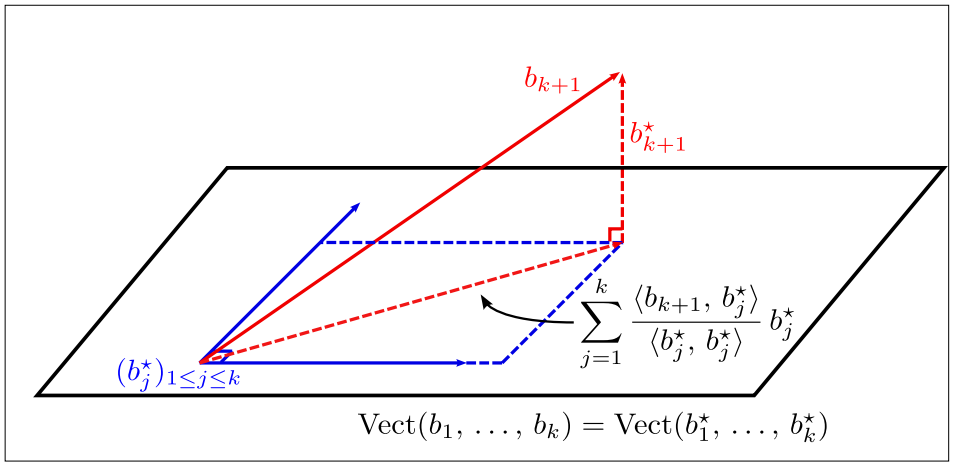



**Figure 2.11 :** Construction du $(k+1)$-ième vecteur de Gram-Schmidt

Une récurrence évidente montre que $\forall\, i$, $\mathrm{Vect}(b_1, \cdots, b_i) = \mathrm{Vect}(b^\star_1, \cdots, b^\star_i)$ et que la famille de Gram-Schmidt est orthogonale.

On notera que la famille de Gram-Schmidt associée à une famille dépend de l'ordre dans lequel sont rangés les vecteurs de la famille. On notera aussi que la famille de Gram-Schmidt associée à la base d'un réseau n'a aucune raison d'être constituée de vecteurs du réseau. En revanche, le premier vecteur de la famille de Gram-Schmidt est toujours égal au premier vecteur de la base du réseau. La figure 2.12 l'illustre.

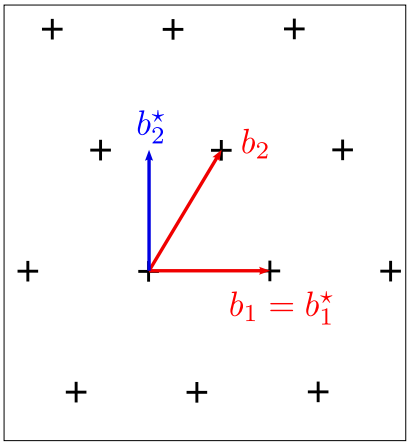



**Figure 2.12 :** Une base d'un réseau et la famille de Gram-Schmidt associée. Le vecteur $b^\star_2$ n'est pas un vecteur du réseau.

La famille de Gram-Schmidt est fortement reliée aux caractéristiques du réseau.

**Proposition 2.45** (Volume d'un réseau)**.** *Soient $B$ et $C$ deux bases d'un réseau. On a*

$$\sqrt{\det({}^tBB)} = \sqrt{\det({}^tCC)}.$$

*Cette quantité dépendant uniquement du réseau (et non de la base considérée) est appelée volume du réseau, ou encore déterminant du réseau [Cas97].*

*Démonstration.* Puisqu'on passe de $B$ à $C$ par une matrice unimodulaire $P$, on a

$$\begin{aligned}\det({}^tCC) &= \det({}^t(BP)(BP)) \\ &= \det({}^tP\,{}^tBBP) \\ &= \det({}^tP)\det({}^tBB)\det(P) \\ &= \det({}^tBB).\end{aligned}$$

□

**Remarque 2.46.** *Soit $B^\star$ la famille de Gram-Schmidt associée à $B$. Alors on montre par le même genre de raisonnement que*

$$\det({}^tBB) = \prod_{j=1}^{p} \left\|b_j^\star\right\|^2.$$

**Proposition 2.47.** *On trouve cette proposition dans [Lov86, lemme 1.2.6] par exemple.*

*Soit $B$ une base d'un réseau et $B^\star$ la famille de Gram-Schmidt associée. Si $\lambda$ désigne la taille d'un plus court vecteur non nul du réseau, on a $\lambda \geq \min_k \|b_k^\star\|$.*

*Démonstration.* Soit $v$ un vecteur du réseau de taille $\lambda$. En écrivant $v$ dans la base $B$, on a $v = a_1b_1 + \cdots + a_pb_p$ avec $a_j \in \mathbb{Z}$. La famille $(b_j^\star/\|b_j^\star\|)$ étant orthonormale, on a

$$\|v\|^2 = \left\| \sum_{j=1}^{p} \left\langle v,\, \frac{b_j^\star}{\left\|b_j^\star\right\|} \right\rangle \frac{b_j^\star}{\left\|b_j^\star\right\|} \right\|^2$$

d'où

$$\lambda^2 = \sum_{j=1}^{p} \frac{\left\langle v,\, b_j^\star \right\rangle^2}{\left\|b_j^\star\right\|^2} \geq \frac{\left\langle v,\, b_{j_0}^\star \right\rangle^2}{\left\|b_{j_0}^\star\right\|^2} \quad \text{pour tout } j_0.$$

En particulier, c'est vrai lorsque $j_0$ est le plus grand indice tel que $a_{j_0} \neq 0$. Or

$$\langle v,\, b_{j_0}^\star \rangle = \sum_{j=1}^{p} a_j \langle b_j,\, b_{j_0}^\star \rangle.$$

Comme $a_j = 0$ pour $j > j_0$ et $\langle b_j,\, b_{j_0}^\star \rangle = 0$ pour $j < j_0$, on a

$$\lambda^2 \geq a_{j_0}^2 \frac{\langle b_{j_0},\, b_{j_0}^\star \rangle^2}{\left\|b_{j_0}^\star\right\|^2}.$$

Or $a_{j_0}$ est un entier non nul, donc $a_{j_0}^2 \geq 1$. Par ailleurs

$$\langle b_{j_0},\, b_{j_0}^\star \rangle = \left\langle \left( b_{j_0}^\star + \sum_{k=1}^{j_0-1} \frac{\langle b_{j_0},\, b_k^\star \rangle}{\langle b_k^\star,\, b_k^\star \rangle} b_k^\star \right),\, b_{j_0}^\star \right\rangle = \langle b_{j_0}^\star,\, b_{j_0}^\star \rangle + \sum_{k=1}^{j_0-1} \frac{\langle b_{j_0},\, b_k^\star \rangle}{\langle b_k^\star,\, b_k^\star \rangle} \langle b_k^\star,\, b_{j_0}^\star \rangle.$$

Or pour $k < j_0$, $\langle b_k^\star,\, b_{j_0}^\star \rangle = 0$. De là, $\langle b_{j_0},\, b_{j_0}^\star \rangle = \langle b_{j_0}^\star,\, b_{j_0}^\star \rangle = \|b_{j_0}^\star\|^2$. On en déduit que $\lambda^2 \geq \|b_{j_0}^\star\|^2 \geq \min_k \|b_k^\star\|^2$ d'où la conclusion. □

**Remarque 2.48.** *Soit $B$ une base d'un réseau $\Gamma$. On note $B^\star$ la famille de Gram-Schmidt associée. Supposons que $b_1^\star$ soit le plus petit vecteur de la famille de Gram-Schmidt. D'après la proposition précédente, $\lambda \geq \|b_1^\star\|$. Or $b_1^\star = b_1$ est un vecteur de $\Gamma$. C'est donc une solution au problème SVP.*

La proposition 2.47 amène la réflexion suivante : le seul vecteur de la famille de Gram-Schmidt sur lequel on ait un réel contrôle est $b_1^\star$ puisque $b_1 = b_1^\star$. Pour imposer que le premier vecteur d'une base soit un plus court vecteur, il suffit donc de s'arranger pour que les vecteurs de la famille de Gram-Schmidt soient de normes croissantes. Puisque SVP est NP-dur, il n'existe probablement pas d'algorithme polynomial permettant, étant donné un réseau $\Gamma$, de trouver une base $B$ pour laquelle la famille de Gram-Schmidt associée est croissante. Mais il est possible, en temps polynomial, de trouver une base $B$ pour laquelle la famille de Gram-Schmidt ne décroît pas trop vite. Nous verrons que cette contrainte peut être réalisée à l'aide du *test de Lovász*.

Nous pouvons à présent dégager le principe de l'algorithme LLL :

1. Étant donnée une base $B$ du réseau, considérer la famille de Gram-Schmidt associée $B^\star$ ; changer la base $B$ pour obtenir une base $B'$ du réseau dont les vecteurs sont proches de ceux de $B^\star$. Cette étape ne modifie pas la famille de Gram-Schmidt associée à la base du réseau.
2. Si un vecteur de la famille de Gram-Schmidt est trop petit par rapport au précédent, changer l'ordre des vecteurs de la base ; ceci change la famille de Gram-Schmidt associée et la rend plus proche des vecteurs du réseaux.
3. Si une permutation a été faite pendant l'étape (2), reprendre à l'étape (1).

À l'issue de l'algorithme, la base est à la fois assez proche de sa famille de Gram-Schmidt (grâce à l'étape 1) et formée de vecteurs assez courts (si l'étape 2 n'a pas donné lieu à une permutation, c'est que les vecteurs de Gram-Schmidt — et donc les vecteurs de la base — sont assez courts).

Plus formellement, l'étape 1 consiste à réduire faiblement la base $B$ et l'étape 2 applique le test de Lovász aux vecteurs de la famille $B^\star$ .

**Définition 2.49** (Réduction faible)**.** On peut trouver cette définition dans [Lov86], par exemple.

Soient $B$ une base d'un réseau et $B^\star$ la famille de Gram-Schmidt associée. $B$ est dite *faiblement réduite* lorsque, pour tout $k$, et pour tout $j < k$, on a

$$\left| \frac{\langle b_k,\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} \right| \leq \frac{1}{2}.$$

On dispose d'un algorithme simple pour transformer une base $B$ en une base faiblement réduite $B'$ de même famille de Gram-Schmidt (nous désignerons par $\lfloor x \rceil$ l'entier le plus proche de $x$ ; si $x$ est à égale distance entre deux entiers, on choisira le plus petit) :

**Proposition 2.50** (Correction de l'algorithme de réduction faible)**.** *On trouve cette proposition dans [Lov86] par exemple.*

*L'algorithme ReduitFaiblement (cf. algorithme 2.2 page suivante), produit une base faiblement réduite.*

*Démonstration.* Si on déroule la boucle d'indice $j$ de l'algorithme, l'expression de $b'_k$ est la suivante :

$$b'_k = \overbrace{\underbrace{\underbrace{b_k}_{t_{k-1}} - \left\lfloor \frac{\langle t_{k-1},\, b_{k-1}^\star \rangle}{\langle b_{k-1}^\star,\, b_{k-1}^\star \rangle} \right\rceil b_{k-1} - \left\lfloor \frac{\langle t_{k-2},\, b_{k-2}^\star \rangle}{\langle b_{k-2}^\star,\, b_{k-2}^\star \rangle} \right\rceil b_{k-2} - \cdots - \left\lfloor \frac{\langle t_1,\, b_1^\star \rangle}{\langle b_1^\star,\, b_1^\star \rangle} \right\rceil b_1}_{t_{k-3}}}^{t_{k-2}}}^{t_1}.$$

**Algorithme :** `ReduitFaiblement`
**Input** : une base $B$ d'un réseau de rang $p$
**Output** : $B'$ faiblement réduite
$B^\star \leftarrow$ `GramSchmidt`$(B)$ ;
$b'_1 \leftarrow b_1$ ;
**for** $k \leftarrow 2$ **to** $p$ **do**
  $t \leftarrow b_k$ ;
  **for** $j \leftarrow k-1$ **downto** $1$ **do**
    $t \leftarrow t - \left\lfloor \frac{\langle t, b_j^\star \rangle}{\langle b_j^\star, b_j^\star \rangle} \right\rceil b_j$ ;
  **end**
  $b'_k \leftarrow t$ ;
**end**
**return** $(b'_1, \ldots, b'_p)$

**Algorithme 2.2 :** Algorithme de réduction faible

Il suit que

$$\langle b'_k,\, b_j^\star \rangle = \left\langle \left( t_j - \left\lfloor \frac{\langle t_j,\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} \right\rceil b_j \right),\, b_j^\star \right\rangle$$

d'où

$$\langle b'_k,\, b_j^\star \rangle = \langle t_j,\, b_j^\star \rangle - \left\lfloor \frac{\langle t_j,\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} \right\rceil \langle b_j,\, b_j^\star \rangle = \langle t_j,\, b_j^\star \rangle - \left\lfloor \frac{\langle t_j,\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} \right\rceil \langle b_j^\star,\, b_j^\star \rangle.$$

Enfin

$$\left| \frac{\langle b'_k,\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} \right| = \left| \frac{\langle t_j,\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} - \left\lfloor \frac{\langle t_j,\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} \right\rceil \right| \leq \frac{1}{2}$$

et l'algorithme produit donc bien une base faiblement réduite. □

**Définition 2.51** (Test de Lovász)**.** Soit $\delta \in ]1/4,\, 1[$. Soient $B$ une base d'un réseau et $B^\star$ la famille de Gram-Schmidt associée. On dit que les vecteurs $b_j$ et $b_{j+1}$ satisfont au test de Lovász de paramètre $\delta$ si

$$\left\| b_{j+1}^\star \right\|^2 \geq \left( \delta - \left( \frac{\langle b_{j+1},\, b_j^\star \rangle}{\langle b_j^\star,\, b_j^\star \rangle} \right)^2 \right) \left\| b_j^\star \right\|^2 .$$

Si on utilise une base faiblement réduite et si tous les vecteurs de la base passent le test de Lovász (on dit alors que la base est LLL-réduite), on voit facilement que, pour tout $j$,

$$\left\| b_{j+1}^\star \right\|^2 \geq (\delta - 1/4) \left\| b_j^\star \right\|^2 .$$

C'est une contrainte de la forme désirée. En particulier, une récurrence évidente montre que pour tout $j$, $\left\| b_j^\star \right\|^2 \geq (\delta - 1/4)^{j-1} \left\| b_1^\star \right\|^2 \geq (\delta - 1/4)^{p-1} \left\| b_1^\star \right\|^2$. En utilisant la proposition 2.47 et le fait que $b_1 = b_1^\star$, on obtient finalement

$$\lambda \geq \sqrt{(\delta - 1/4)^{p-1}}\, \|b_1\| ,$$

où $\lambda$ désigne toujours la taille d'un plus court vecteur non nul dans le réseau.

```
1  Algorithme : LLL
   Input : une base B d'un réseau
   Output : une base LLL-réduite
2  while true do
3  |  B* ← GramSchmidt(B) ;
4  |  B' ← ReduitFaiblement(B) ;
5  |  if ∀ j, b_j et b_{j+1} passent le test de Lovász then
6  |  |  goto 14 ;
7  |  else
8  |  |  j ← indice du premier échec au test de Lovász ;
9  |  |  t ← b_j ;
10 |  |  b_j ← b_{j+1} ;
11 |  |  b_{j+1} ← t ;
12 |  end
13 end
14 return B';
```

**Algorithme 2.3 :** Algorithme LLL

Par exemple, pour $\delta = 3/4$ (qui est la valeur du paramètre couramment choisie pour exécuter LLL), le premier vecteur d'une base LLL est de norme inférieure à $2^{(p-1)/2}$ fois la norme du plus court vecteur non nul.

L'algorithme LLL consiste donc à effectuer en boucle successivement une réduction faible, puis le test de Lovász sur chaque paire de vecteurs consécutifs de la base. Si le test échoue sur une paire $(j,\ j+1)$, on échange les vecteurs $b_j$ et $b_{j+1}$ dans la base et on recommence. Ce qui n'est pas évident, c'est que ce processus s'arrête. Sans entrer dans les détails de la preuve (qui est assez technique et que l'on pourra consulter par exemple dans [Coh93, Chap. 2.6]), disons simplement que l'algorithme s'arrête, et en temps polynomial qui plus est. Pour cela, on montre que, à chaque fois que le test de Lovász échoue et qu'une permutation est réalisée, le produit des volumes fondamentaux des réseaux de base $(b_1)$, $(b_1, b_2)$, $\ldots$ , $(b_1, b_2, \cdots, b_p)$ diminue d'un facteur au moins égal à $1/\delta$. On peut montrer que ce produit est borné inférieurement par une constante strictement positive (indépendante des $b_j$) et il ne peut donc pas diminuer indéfiniment.

### Problème du plus proche vecteur

Le problème du plus proche vecteur (ou encore CVP, de l'anglais *closest vector problem*) est la variante affine du problème du plus court vecteur. SVP consiste à trouver, dans un réseau $\Gamma$, le point non nul le plus proche de 0. De la même façon, CVP consiste à trouver, dans $\Gamma$, le point le plus proche d'un vecteur $v$ donné : étant donné un réseau de base $B$ d'un espace vectoriel $E$, et un vecteur $v$ de $E$, on cherche un vecteur $t$ dans le réseau minimisant $\|v - t\|$.

Nous pouvons faire l'hypothèse que le réseau est de rang plein, c'est-à-dire que le rang du réseau est égal à la dimension de l'espace sous-jacent. En effet, si $\Gamma$ n'est pas de rang plein, notons $\pi(v)$ la projection orthogonale de $v$ sur l'espace vectoriel engendré par $B$. Le théorème de Pythagore indique que, pour $t \in \mathrm{Vect}(B)$,

$$\|v - t\|^2 = \|v - \pi(v)\|^2 + \|\pi(v) - t\|^2.$$

Par conséquent $t$ minimise $\|v-t\|$ si et seulement s'il minimise aussi $\|\pi(v)-t\|$. Quitte à remplacer $v$ par $\pi(v)$, on pourra donc supposer que $v$ est dans l'espace vectoriel engendré par $B$.

Tout comme SVP (et pour les mêmes raisons), CVP admet au moins une solution, celle-ci n'étant pas nécessairement unique. Nous avons vu qu'une base LLL-réduite fournissait une approximation de SVP. On peut aussi l'utiliser pour obtenir une approximation du CVP. Pour cela, on utilise l'algorithme de Babai [Bab86] (cf. algorithme 2.4). Cet algorithme, qui s'inspire fortement de l'algorithme de réduction faible, assure que les coefficients de l'écriture de $v - t$ dans la base $B^\star$ sont inférieurs à $1/2$ en valeur absolue.

On a alors

$$\|v - t\|^2 \leq \frac{1}{4} \sum_{j=1}^{p} \|b_j^\star\|^2 \leq \frac{1}{4} \sum_{j=1}^{p} \frac{1}{(\delta - 1/4)^{p-j}} \left\|b_p^\star\right\|^2 \tag{2.8}$$

car la base est LLL-réduite. De là, en exprimant la somme d'une suite géométrique et en simplifiant, on obtient

$$\|v - t\|^2 \leq \frac{1}{4} (\delta - 1/4) \frac{(\delta - 1/4)^{-p} - 1}{5/4 - \delta} \left\|b_p^\star\right\|^2.$$

Finalement, on obtient la majoration

$$\|v - t\|^2 \leq \left(\frac{\delta - 1/4}{5/4 - \delta}\right) (\delta - 1/4)^{-p} \frac{\left\|b_p^\star\right\|^2}{4}.$$

**Algorithme :** `Babai`
**Input** : une base LLL-réduite $B$, un vecteur $v$
**Output** : une approximation du CVP de $v$
$t \leftarrow v$ ;
**for** $j \leftarrow p$ **downto** 1 **do**
  $t \leftarrow t - \left\lfloor \frac{\langle t, b_j^\star \rangle}{\langle b_j^\star, b_j^\star \rangle} \right\rceil b_j$ ;
**end**
**return** $v - t$ ;

**Algorithme 2.4 :** Algorithme de Babai

Considérons une solution exacte $\tau$ de CVP et supposons par souci de simplicité que les composantes suivant $b_j$ de $\tau$ et de $t$ sont différentes (en cas d'égalité, on considérerait le plus grand indice $s$ pour lequel les composantes sont différentes). Elles diffèrent d'une certaine valeur entière $k$. Alors, les composantes $\alpha_p$ et $a_p$ de $v - \tau$ et de $v - t$ suivant $b_p^\star$ diffèrent aussi de $k$. Or $a_p \in [-\frac{1}{2}, \frac{1}{2}[$, donc $|\alpha_p| \geq 1/2$.

Il suit que $\|v - \tau\|^2 \geq \alpha_p^2 \left\|b_p^\star\right\|^2 \geq \frac{\left\|b_p^\star\right\|^2}{4}$. De là, on déduit finalement que

$$\|v - t\|^2 \leq \left(\frac{\delta - 1/4}{5/4 - \delta}\right) (\delta - 1/4)^{-p} \|v - \tau\|^2.$$

Par exemple, pour $\delta = 3/4$, l'approximation obtenue est au pire $2^{p/2}$ fois plus grande que l'optimal.

**Remarque 2.52.** *Soit $B$ une base d'un réseau $\Gamma$. Soit $B^\star$ la famille de Gram-Schmidt associée. On suppose que le vecteur $b_p^\star$ est le plus grand de la famille de Gram-Schmidt. Alors, dans l'inégalité (2.8), au lieu de majorer par la somme géométrique, on peut faire la majoration*

$$\sum_{j=1}^{p} \|b_j^\star\|^2 \leq p \|b_p^\star\|^2.$$

*Dans ce cas, la qualité du vecteur renvoyé est donc bien meilleure :*

$$\|v - t\| \leq \sqrt{p}\, \|v - \tau\|.$$

### 2.3.7 Algorithme FPminimax

Nous pouvons à présent revenir coller les morceaux qui permettent de résoudre notre problème initial. On se donne $n+1$ fonctions de base $\varphi_0, \ldots, \varphi_n$ continues sur un intervalle $[a, b]$ et une fonction continue à approcher $g$. En outre, on se donne une liste de $n+1$ formats (virgule fixe, virgule flottante ou expansions de flottants). On cherche à obtenir rapidement un « polynôme » $p = \sum_{j=0}^{n} a_j \varphi_j$ dont les coefficients soient représentables sur les formats demandés et tel que $\|p - g\|_\infty$ soit petit.

Voici tout d'abord le schéma général de l'algorithme (nous le détaillerons ensuite point par point) :

1. Avant tout, on calcule le « polynôme » de meilleure approximation à coefficients réels $p^\star$ (voir sections 2.2.5 et 2.2.6).
2. On se ramène au cas où les coefficients sont des entiers.
3. On choisit (au moins) $n+1$ points (ou éventuellement plus) $z_0, \ldots, z_n$ : on cherche à interpoler $p^\star$ aux points $z_i$ de manière approchée.
4. Formellement, il s'agit d'un problème de plus proche vecteur dans un réseau euclidien $\Gamma$ : on le résout de façon approchée avec l'algorithme de Babai.

#### Heuristique pour se ramener au cas entier

Nous avons vu en section 2.3.3 une méthode rigoureuse pour nous ramener au cas où les coefficients sont entiers : nous avons construit un polytope $\mathcal{P}$ (voir page 104 et suivantes) et nous nous sommes servis des projections orthogonales du polytope $\mathcal{P}$ pour encadrer la valeur de chaque coefficient.

Ceci nous a permis de déterminer la partie haute des expansions de flottants (voir exemple 2.32) : ainsi nous nous sommes ramenés au cas où les coefficients étaient tous des nombres en virgule fixe ou en virgule flottante.

De même, les encadrements fournis par les projections orthogonales nous ont permis de déterminer les exposants des nombres flottants (voir exemple 2.35). Nous nous sommes ainsi ramenés au cas où les coefficients étaient tous en virgule fixe. En remplaçant les fonctions de base $\varphi_j$ par des fonctions de la forme $\varphi_j / 2^{E_j}$, on est alors ramené au cas où les coefficients inconnus sont des entiers.

Cette méthode présente l'avantage de la rigueur. Cependant, elle est un peu lourde à mettre en œuvre. En effet, il faut :

— choisir avec soin les points $x_i$ qui servent à construire le polytope $\mathcal{P}$ ;
— choisir une erreur $\varepsilon_{\text{cible}}$ suffisamment petite pour que les projections orthogonales fournissent des encadrements fins ;
— et réaliser les $n+1$ projections (soit $2n+2$ appels à l'algorithme du simplexe).

Si, comme c'est le cas désormais, on ne cherche pas à obtenir la liste des polynômes optimaux mais seulement à avoir *rapidement* de bons polynômes, on peut accepter de mettre en place une heuristique pour se ramener au cas de coefficients entiers. Si, au bout du compte, on obtient un polynôme $p$ de bonne qualité et dont les coefficients respectent les formats, il n'y a pas lieu de chercher à être plus rigoureux. D'ailleurs, si $p$ est vraiment de bonne qualité, il est toujours possible d'utiliser la méthode des projections en prenant $\varepsilon_{\text{cible}} = \|p - g\|_\infty$ et en choisissant les extrema de $p - g$ pour points $x_i$ (à la manière de ce que nous avons fait dans l'exemple 2.35) :

les encadrements obtenus sont alors généralement de bonne qualité et permettent de conclure rigoureusement *a posteriori*.

**Expansions de flottants.** Pour gérer les expansions de flottants, on utilise simplement la remarque suivante : tout nombre flottant sur 107 bits est représentable sur un double-`double` et tout nombre flottant de 161 bits est représentable sur un triple-`double`. La réciproque est fausse : par exemple $1 + (2^{53} - 1)\,2^{-115}$ est un double-`double` mais n'est pas représentable sur moins de 116 bits. Ceci correspond à un « trou » entre les deux termes du double-`double`. Plus précisément, considérons un double-`double` $x_h + x_\ell$ (avec $|x_h| \geq |x_\ell|$) : si le bit de poids faible de $x_h$ est plus grand que le bit de poids fort de $x_\ell$, le développement binaire de $x = x_h + x_\ell$ contient une série de 0 consécutifs (ou de 1 consécutifs si $x_h$ et $x_\ell$ sont de signes différents). Dans notre exemple :

$$1 + \frac{2^{53} - 1}{2^{115}} = \underbrace{\boxed{1,0\cdots0}}_{x_h}\ \underbrace{\boxed{0\cdots0}}_{10\text{ zéros}}\ \underbrace{\boxed{11\cdots1}}_{x_\ell}.$$

Si un coefficient $a_j$ doit être représenté sur un double-`double`, on peut souvent faire l'hypothèse que le trou entre les deux doubles sera relativement petit. En choisissant de représenter $a_j$ sur un nombre flottant de 107 bits, on perd donc un peu en précision, mais sans doute pas trop.

**Détermination des exposants.** Pour déterminer les exposants, nous proposons l'heuristique suivante. On peut penser que $p$ n'est pas très différent de $p^\star$ (c'est l'hypothèse que nous faisons lorsque nous cherchons $p$ comme polynôme interpolateur approché de $p^\star$). Par conséquent, nous pouvons supposer que le coefficient $a_j$ inconnu sera assez proche du coefficient $a_j^\star$ de $p^\star$. Ainsi, l'exposant de $e_j$ est probablement le même que l'exposant de $a_j$. Ceci donne donc un premier jeu d'exposants $e_0, \ldots, e_n$. On est alors ramené à un problème à coefficients entiers que l'on résout à l'aide de l'algorithme de Babai.

On obtient ainsi un polynôme $p$. Il se peut que nous nous soyons trompés : certains coefficients $a_j$ de $p$ peuvent alors avoir un exposant $e_j$ différent de celui qu'on avait initialement deviné. Dans ce cas, on peut imaginer que les exposants $e_j$ sont corrects et recommencer avec ce nouveau jeu d'exposants. Et ainsi de suite jusqu'à ce que les exposants se stabilisent. En pratique, cette stratégie s'arrête en une ou deux étapes la plupart du temps.

> **Remarque 2.53.** *Les trous dans les expansions peuvent être traités avec la même heuristique. Supposons qu'un coefficient $a_j$ doive être représenté sur un double-`double`. Si on fait l'hypothèse suivant laquelle $a_j \simeq a_j^\star$, on peut regarder s'il y a un trou dans $a_j^\star$ après le $53^e$ bit et quelle est sa taille (pour reprendre notre exemple, supposons un trou de taille 10). On peut alors chercher $a_j$ comme un nombre flottant sur 116 bits.*
> *Si l'hypothèse était correcte, on va effectivement obtenir un coefficient $a_j$ sur 116 bits faisant apparaître un trou de 10 bits. Il sera donc représentable exactement par un double-`double`.*

### Choix des points d'interpolation

Si la condition de Haar est remplie, le théorème d'alternance (théorème 2.5 page 82) est vérifié et $p^\star - g$ oscille donc au moins $n + 2$ fois entre ses valeurs extrêmes. En appliquant le théorème des valeurs intermédiaires entre deux extrema successifs, on en déduit que $p^\star - g$ a au moins $n + 1$ zéros dans l'intervalle $[a, b]$. Notons $z_i$ ces zéros : alors par définition $p^\star(z_i) = g(z_i)$. Autrement dit, $p^\star$ interpole la fonction $g$ aux points $z_i$. Il est alors raisonnable de prendre ces points-là pour la construction du réseau $\Gamma$ : en effet, une solution $p$ du problème d'interpolation approchée vérifiera

alors pour tout $i$, $p(z_i) \simeq p^\star(z_i) = g(z_i)$. Autrement dit, $p$ sera proche non seulement de $p^\star$ mais aussi de $g$, ce que nous voulons en fin de compte.

Évidemment, lorsque la condition de Haar n'est pas remplie, $p^\star - g$ peut ne pas avoir $n+1$ racines. Dans ce cas, il semble raisonnable d'utiliser des points bien répartis dans l'intervalle. Les points correspondants aux zéros du polynôme de Tchebychev de degré $n+1$ sont généralement un bon choix pour faire de l'interpolation [Che82, Chap. 3, Sec. 2] :

$$z_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{(n-i+1/2)\,\pi}{n+1}\right) \quad i \in [\![0,\, n]\!]\,. \tag{2.9}$$

**Construction du réseau et résolution approchée**

Une fois qu'on s'est ramené à un problème à coefficients entiers et qu'on a choisi des points $z_i$, il nous reste à interpoler $p^\star$ de manière approchée aux points $z_i$. Formellement, en posant

$$b_j = {}^t(\varphi_j(z_0),\, \ldots,\, \varphi_j(z_n)) \quad \text{et} \quad v = {}^t(p^\star(z_0),\, \ldots,\, p^\star(z_n)),$$

cela revient à chercher des coefficients $(a_0,\, \ldots,\, a_n) \in \mathbb{Z}^{n+1}$ pour minimiser

$$\left\| \sum_{j=0}^{n} a_j\, b_j - v \right\|.$$

En d'autres termes, on cherche, dans le réseau $\Gamma$ de base $B = (b_0,\, \ldots,\, b_n)$, le vecteur le plus proche de $v$. Pour résoudre ce problème, nous réduisons la base $B$ en une base $B'$ à l'aide de l'algorithme LLL, puis nous utilisons l'algorithme de Babai avec la base $B'$ pour trouver une approximation $t$ du CVP. Ceci nous donne l'expression du vecteur $t$ dans la base $B'$ : $t = \sum_{j=0}^{n} a'_j\, b'_j$. Pour résoudre le problème initial, il suffit alors de changer de base et d'écrire $t$ dans la base $B$ : $t = \sum_{j=0}^{n} a_j\, b_j$. Les coefficients $a_j$ ainsi obtenus fournissent une solution approchée au problème d'interpolation.

Évidemment, cet algorithme est uniquement heuristique ; nous commettons deux approximations :

— d'abord nous remplaçons le problème d'approximation en norme sup par un problème d'interpolation approchée au sens de la norme euclidienne. Une solution approchée au problème d'interpolation fournit en général un bon polynôme, mais il n'a aucune raison d'être optimal au sens de la norme sup ;

— ensuite, nous ne résolvons même pas exactement le problème CVP : *a priori*, l'algorithme de Babai ne fournit qu'une solution approchée au problème CVP.

En pratique, nous avons remarqué que l'algorithme LLL se comporte remarquablement bien sur les réseaux liés aux problèmes d'approximation. Pour un réseau quelconque, si on note $B^\star$ la famille de Gram-Schmidt associée à une base LLL-réduite, on observe en général une décroissance des $\|b_i^\star\|$ :

$$\|b_0^\star\| \geq \|b_1^\star\| \geq \cdots \geq \|b_n^\star\|.$$

L'algorithme LLL est là pour limiter cette décroissance à une vitesse au plus géométrique.

Sur les réseaux que nous manipulons dans le cadre de l'interpolation approchée, ce comportement est complètement inversé : les $\|b_i^\star\|$ ont tendance à croître. Ils croissent même souvent à une vitesse exponentielle. La figure 2.13 représente un cas typique de la taille des $\|b_i^\star\|$ en fonction de $i$. Elle correspond à la base réduite qu'on obtient pour l'exemple 2.26 (approcher la fonction $\log_2(1+2^{-x})$ en erreur absolue par un polynôme de degré 6 sur $[0,\, 1]$ avec des coefficients en simple précision). Le comportement qu'on observe sur cet exemple est représentatif de tous les cas que nous avons traités. Nous n'avons pas, à l'heure actuelle, d'argument pour expliquer cette croissance

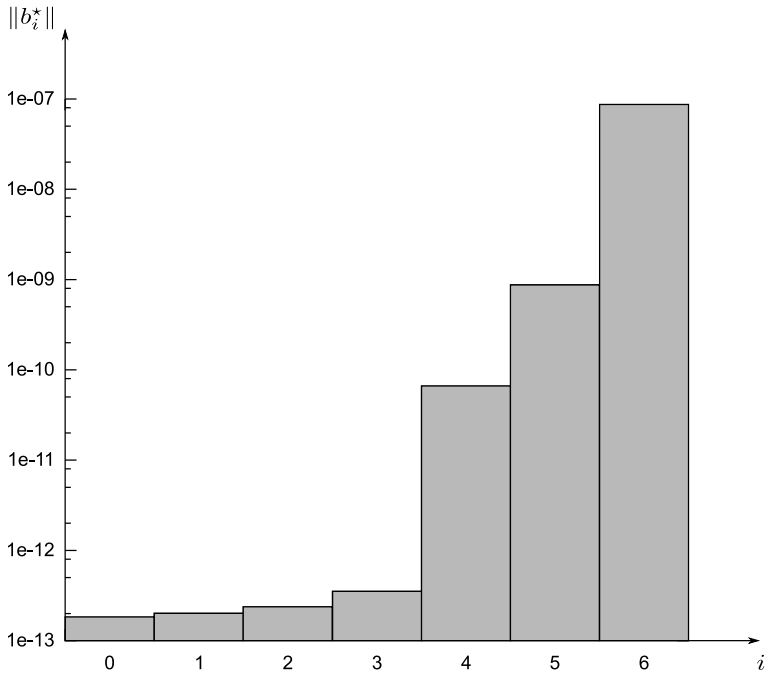



**Figure 2.13 :** Croissance des $\|b_i^\star\|$ en fonction de $i$ dans l'exemple 2.26. On notera que l'échelle est logarithmique : la croissance est donc très rapide (au moins pour les dernières valeurs de $i$).

de $\|b_i^\star\|$ en fonction de $i$. Ce phénomène singulier a une conséquence heureuse : l'algorithme de Babai donne une très bonne approximation du CVP lorsqu'on l'utilise sur ce genre de bases (voir remarque 2.52).

### Rayonner autour du CVP

Même si, sur les réseaux qui proviennent de notre problème, l'algorithme de Babai donne une très bonne approximation du CVP, il demeure que la solution $p$ du problème d'interpolation approchée ne constitue pas nécessairement une solution optimale au problème d'approximation en norme sup. Cependant, on s'attend (et c'est ce qu'on observe en pratique) à ce que $p$ soit d'assez bonne qualité.

On rappelle qu'on note $v$ le vecteur correspondant aux $p^\star(z_i)$ et $t$ la solution du CVP. Les points du réseau qui sont proches de $t$ sont également proches de $v$ et les polynômes associés constituent aussi de bonnes approximations de $g$. Il peut donc être intéressant de se promener, dans le réseau $\Gamma$, autour du point $t$ : peut-être les polynômes correspondants sont-ils meilleurs en norme sup. Comme la base LLL est formée de vecteurs assez courts et assez orthogonaux, on peut l'utiliser pour explorer le voisinage de $t$. Ainsi, les $t \pm b_j$ fournissent d'autres polynômes $p$ pour lesquels $\|p - g\|_\infty$ est petit.

**Exemple 2.54.** *Considérons l'exemple 2.26. On applique la méthode heuristique pour trouver un bon polynôme d'approximation en suivant le schéma général décrit en page 125 : calcul de $p^\star$, détermination heuristique des exposants en utilisant les coefficients de $p^\star$, choix des points $z_i$ tels que $p^\star(z_i) = g(z_i)$, construction du réseau $\Gamma$ associé, réduction LLL et algorithme de Babai.*
*Notons $t$ le vecteur trouvé par l'algorithme de Babai et $(b'_0, \ldots, b'_6)$ la base réduite. Les coefficients de $t$ dans la base des $b_j$ fournissent le polynôme $p_0$ défini dans l'exemple 2.35 page 108. Ce polynôme donne une erreur d'environ $10,33\,e{-}10$.*
*En calculant les 14 polynômes correspondants aux vecteurs $(t \pm b'_j)$ et en mesurant la norme sup de l'erreur correspondante, on s'aperçoit que le polynôme correspondant à $t - b'_5$*

*fournit une erreur d'environ* $10,24\,e{-}10$*, ce qui est légèrement mieux.*

### Implantation dans Sollya

L'approche heuristique utilisant les réseaux euclidiens a été implémentée dans Sollya. Pour effectuer la réduction du réseau, nous utilisons la bibliothèque FpLLL [9] écrite par Damien Stehlé. La commande correspondante dans Sollya s'appelle `fpminimax`. Comme toutes les autres commandes de Sollya, sa syntaxe est entièrement documentée dans le manuel de Sollya [CJJL] et le lecteur intéressé pourra s'y reporter. La commande a été conçue dans le double but suivant :

— permettre une utilisation aisée et rapide avec des heuristiques par défaut correspondant à ce qui nous semble être l'usage le plus courant ;
— mais permettre aussi à ceux qui connaissent bien l'algorithme de jouer sur tous les paramètres.

Ainsi par défaut, `fpminimax` cherche à optimiser l'erreur relative : il suffit de donner une fonction $g$, un degré $n$ (ou une liste d'entiers $(\alpha_0, \ldots, \alpha_n)$ si on veut chercher un polynôme dans la base des $\varphi_j(x) = x^{\alpha_j}$), un intervalle $[a, b]$ et une liste d'entiers $(t_0, \ldots, t_n)$ indiquant la précision en bits voulue pour chaque coefficient. La commande renvoie un polynôme $p$ de degré inférieur ou égal à $n$ dont le coefficient $a_j$ est un nombre flottant représentable sur $t_j$ bits.

**Exemple 2.55.** *Pour traiter l'exemple 2.27 avec Sollya, il suffit de reproduire la session suivante. La commande* `fpminimax` *renvoie le polynôme $p$ indiqué dans le tableau 2.7 page 103.*

```
> p=fpminimax(cos(pi*x), [|0,2,4,6|], [|53...|], [0;1/256]);
Warning in Remez: a slower algorithm is used for this step
> dirtyinfnorm(p-cos(pi*x), [0;1/256]);
3.3380575760874489924505801975622174414061312388058e-22
```

*Sollya calcule automatiquement le polynôme de meilleure approximation de $g$, trouve les exposants de façon heuristique, choisit une liste de points pour l'interpolation approchée, construit le réseau associé, lance l'algorithme LLL, trouve une approximation du plus proche vecteur, fait le changement de base entre la base réduite et la base initiale, vérifie que le polynôme reconstruit a des coefficients avec les bons exposants, etc.*

Bien sûr, il est très facile d'optimiser l'erreur absolue plutôt que relative. Ainsi, on peut traiter l'exemple 2.26 de la façon suivante :

```
> p=fpminimax(log2(1+2^(-x)), 6, [|24...|], [0;1], absolute);
> dirtyinfnorm(p-log2(1+2^(-x)), [0;1]);
1.033080719826012561327503164097540733527489065443 85e-9
```

On trouve le polynôme $p_0$ présenté dans l'exemple 2.35 page 108. Son erreur est bien environ $10,33\,e{-}10$.

### Récapitulatif de l'exemple fil rouge

Reprenons une dernière fois notre exemple fil rouge (exemple 2.29). Au départ, nous ne connaissions que l'erreur du minimax $p^\star$ et celle du polynôme $\widehat{p}$ obtenu en arrondissant les coefficients de $p^\star$.

9. distribuée sous licence LGPL et disponible à l'adresse `http://perso.ens-lyon.fr/damien.stehle/#software`

```
> g = (2^x-1)/x;
> pstar=remez(g, 9, [-1/16; 1/16]);
> pchap= doubledouble(coeff(pstar,0));
> for i from 1 to 9 do pchap = pchap + doubleextended(coeff(pstar,i))*x^i;
>
> dirtyinfnorm(pstar-g, [-1/16;1/16]);
7.89716884508706067206893297336029437368165243606 45e-25
> dirtyinfnorm(pchap-g, [-1/16;1/16]);
4.03529385422827255126290837380286198858890787063 86e-22
```

Puisqu'on a déjà calculé $p^\star$, on peut le passer en argument à `fpminimax` qui n'a donc pas besoin de le recalculer. En utilisant `fpminimax`, on peut alors obtenir un premier polynôme qui fournit une erreur bien meilleure.

```
> p=fpminimax(g, 9, [|DD,DE...|], [-1/16;1/16],absolute,default,default,pstar);
> dirtyinfnorm(p-g,[-1/16;1/16]);
5.32604267159612448755509874029040571177271086192 55e-23
```

Mais on peut faire mieux : en utilisant les projections orthogonales, on montre que la partie haute du coefficient constant est fixée (exemple 2.32). De plus le coefficient d'ordre 1 n'a qu'une seule valeur possible (exemple 2.33). On peut en tenir compte en indiquant à `fpminimax` qu'on cherche un polynôme avec une partie contrainte. Plus précisément, on veut que $p$ soit de la forme

$$p(x) = \frac{6243314768165359}{2^{53}} + a_0 + \frac{17725587574382949699}{2^{66}}\, x + a_2\, x^2 + \cdots + a_9\, x^9$$

où $a_0$ est un `double` et $a_2$, …, $a_9$ sont des `double` étendus.

```
> p2=fpminimax(g, [|0,2,...,9|], [|D,DE...|], [-1/16;1/16], absolute,62433147681
65359*2^(-53) + 17725587574382949699*2^(-66)*x);
> dirtyinfnorm(p2-g,[-1/16;1/16]);
4.44910119598993162533763479939651028765260171839 82e-23
```

L'utilisateur expérimenté peut également utiliser `fpminimax` en choisissant lui-même les exposants (ce qui correspond à des coefficients en virgule fixe), ou encore en fixant lui-même la liste des points $z_i$ d'interpolation.

## 2.4 E-méthode et approximation rationnelle contrainte

Jusqu'à présent, nous nous sommes essentiellement concentrés sur la problématique de l'approximation polynomiale. En effet, comme nous l'avons vu, les polynômes sont un choix de prédilection pour l'évaluation de fonctions numériques, puisqu'ils peuvent être évalués en n'utilisant que des additions et des multiplications, qui sont les opérations les plus rapides sur les processeurs actuels. De plus, l'exploitation du parallélisme dans l'évaluation de polynômes a été largement étudiée (voir par exemple [Knu98, Chap. 4.6.4] ou [Rev06]).

Cependant, si on cherche à implémenter une fonction numérique directement sous forme d'un opérateur matériel, les choses changent légèrement. En effet, il est possible alors de dessiner un circuit spécifique pour évaluer la fonction d'approximation. Cela ouvre la voie à l'utilisation d'approximations moins conventionnelles, telles que certaines fractions rationnelles. En effet, M. Ercegovac a proposé en 1975 [Erc75, Erc77] une méthode permettant l'évaluation efficace de *certaines*

fractions rationnelles à l'aide d'un circuit *ad hoc* qui fournit un bit du résultat écrit en virgule fixe dans un codage redondant, à chaque cycle d'horloge. Cette méthode, qui présente une ressemblance avec l'algorithme de division SRT, permet d'évaluer la fraction rationnelle en utilisant uniquement des additions. Ainsi, il devient concevable d'utiliser non seulement des polynômes, mais aussi des fractions rationnelles pour approcher la fonction $f$ que l'on désire implanter.

### 2.4.1 E-méthode

Pour comprendre le fonctionnement de la méthode d'Ercegovac (appelée E-méthode), imaginons que nous voulions évaluer la fraction

$$R(x) = \frac{p_0 + \cdots + p_n x^n}{1 + q_1 x + \cdots + q_n x^n} = \frac{p(x)}{1 + q(x)}.$$

Cette situation est tout à fait générale :

- si le terme constant du dénominateur n'est pas nul, on peut normaliser la fraction en multipliant numérateur et dénominateur par une même constante pour obtenir une fraction de la forme de $R$ ;
- si le terme constant est nul, on factorise la plus grande puissance de $x$ possible au numérateur et au dénominateur et on obtient alors un produit de la forme $x^k R(x)$ où $k \in \mathbb{Z}$. En pratique, on évaluera donc $R(x)$ et $x^k$ en parallèle et on effectuera une multiplication à la fin ;
- si le numérateur et le dénominateur ne sont pas de même degré, on peut ajouter des coefficients $p_i$ ou $q_i$ nuls pour se ramener dans la situation de la fraction $R$.

Notons $y = R(x)$ : c'est donc la valeur que nous voulons calculer. Nous allons exprimer $y$ comme un point fixe. Tout d'abord, on réduit au même dénominateur : $y(1 + q(x)) = p(x)$ soit encore $y = p(x) - yq(x)$. Nous pouvons donc écrire $y$ comme un polynôme en $x$ dont les coefficients dépendent de $y$ :

$$y = p_0 + \sum_{i=1}^{n} (p_i - q_i y) \cdot x^i = \sum_{i=0}^{n} \alpha_i x^i.$$

Évaluer le polynôme $\sum_{i=0}^{n} \alpha_i \, x^i$ par le schéma de Horner revient à écrire la récurrence

$$\left\{ \begin{array}{rcccl} y_n & = & \alpha_n & = & p_n - q_n y \\ y_i & = & \alpha_i + x y_{i+1} & = & (p_i - q_i y) + x y_{i+1} \\ y_0 & = & \alpha_0 + x y_1 & = & p_0 + x y_1 \\ y & = & y_0 \, . & & \end{array} \right. \tag{2.10}$$

Nous allons chercher à calculer les $y_i$ en base 2 en utilisant l'ensemble de chiffres redondant $\{-1, 0, 1\}$. Quitte à modifier la position de la virgule, on peut donc écrire

$$y_i = d_i^{(0)}, d_i^{(1)} d_i^{(2)} \ldots \quad (d_i^{(j)} \in \{-1, 0, 1\}).$$

L'écriture de $y_i$ n'est alors pas unique. Nous voulons calculer les chiffres de $y_i$ un à un, de telle sorte que le développement partiel $y_i^{(j)} = d_i^{(0)}, d_i^{(1)} \cdots d_i^{(j)}$ obtenu à l'étape $j$ puisse toujours être complété pour former un développement correct de $y_i$. Les $y_i^{(j)}$ ne vérifient évidemment pas exactement le système linéaire (2.10). On note $\varepsilon_i^{(j)}$ l'erreur correspondante :

$$\left\{ \begin{array}{rcl} y_n^{(j)} - (p_n - q_n \, y_0^{(j)}) & = & \varepsilon_n^{(j)} \\ y_i^{(j)} - \left( (p_i - q_i \, y_0^{(j)}) + x \, y_{i+1}^{(j)} \right) & = & \varepsilon_i^{(j)} \\ y_0^{(j)} - (p_0 + x \, y_1^{(j)}) & = & \varepsilon_0^{(j)}. \end{array} \right. \tag{2.11}$$

Imaginons que nous avons calculé les $y_i^{(j-1)}$. Nous cherchons donc maintenant à calculer $y_i^{(j)}$ ou, en d'autres termes, à déterminer une valeur possible $d_i^{(j)}$ pour le $j$-ème chiffre de $y_i$. Ce chiffre représente la différence (à un facteur $2^j$ près) entre $y_i^{(j-1)}$ et $y_i^{(j)}$ :

$$\begin{array}{rcl} y_i^{(j)} - y_i^{(j-1)} & = & \begin{array}{r} d_i^{(0)}, d_i^{(1)} \cdots d_i^{(j-1)} d_i^{(j)} \\ \hline - d_i^{(0)}, d_i^{(1)} \cdots d_i^{(j-1)} 0 \end{array} \\ & = & \begin{array}{r} 0 \,.\, 0 \quad \cdots \quad 0 \quad d_i^{(j)} \end{array} \\ & = & d_i^{(j)}/2^j. \end{array}$$

En faisant la soustraction correspondante dans le système (2.11), on fait donc apparaître une relation entre les $d_i^{(j)}$ et les $\varepsilon_i^j$ :

$$\left\{ \begin{array}{rcl} d_n^{(j)} + q_n\, d_0^{(j)} & = & 2^j\, (\varepsilon_n^{(j)} - \varepsilon_n^{(j-1)}) \\ d_i^{(j)} + (q_i\, d_0^{(j)} - x\, d_{i+1}^{(j)}) & = & 2^j\, (\varepsilon_i^{(j)} - \varepsilon_i^{(j-1)}) \\ d_0^{(j)} - x\, d_1^{(j)} & = & 2^j\, (\varepsilon_0^{(j)} - \varepsilon_0^{(j-1)}). \end{array} \right. \tag{2.12}$$

Pour assurer la convergence, il nous suffit de faire en sorte que, pour tout $i$, $\varepsilon_i^{(j)}$ tende vers 0 lorsque $j$ augmente. Pour simplifier les équations, nous introduisons le résidu $w_i^{(j)} = -2^j \varepsilon_i^{(j-1)}$. Il nous suffit de faire en sorte que $w_i^{(j)}$ reste borné. D'après (2.12), on a

$$\left\{ \begin{array}{rcl} w_n^{(j+1)} & = & 2\, (\, w_n^{(j)} - d_n^{(j)} - q_n\, d_0^{(j)}\, ) \\ w_i^{(j+1)} & = & 2\, (\, w_i^{(j)} - d_i^{(j)} - (q_i\, d_0^{(j)} - x\, d_{i+1}^{(j)})\, ) \\ w_0^{(j+1)} & = & 2\, (\, w_0^{(j)} - d_0^{(j)} + x\, d_1^{(j)}\, ). \end{array} \right.$$

Notons que puisque $d_0^{(j)}$, $d_{i+1}^{(j)}$ et $d_1^{(j)}$ sont des éléments de l'ensemble $\{-1, 0, 1\}$, les grandeurs $q_n\, d_0^{(j)}$, $(q_i\, d_0^{(j)} - x\, d_{i+1}^{(j)})$ et $x\, d_1^{(j)}$ sont naturellement bornées. Il faut donc choisir $d_i^{(j)}$ de telle sorte que $w_i^{(j)} - d_i^{(j)}$ soit petit. On en déduit une règle pour choisir $d_i^{(j)}$.

| Choisir pour $d_i^{(j)}$ l'entier le plus proche de $w_i^{(j)}$. |
|---|

Pour que cette règle puisse être appliquée, il faut que $|w_i^{(j)}|$ soit inférieur à $3/2$. On a alors $|d_i^{(j)} - w_i^{(j)}| \leq 1/2$ et il suffit donc que $|q_n\, d_0^{(j)}| \leq 1/4$, $|q_i\, d_0^{(j)} - x\, d_{i+1}^{(j)}| \leq 1/4$ et $|x\, d_1^{(j)}| \leq 1/4$. Nous en déduisons des conditions suffisantes pour que la méthode puisse fonctionner.

$$\begin{array}{|r|} \hline |q_n| \leq 1/4 \\ \forall i \in [\![1,\, n-1]\!]\,,\, |q_i| + |x| \leq 1/4 \\ |x| \leq 1/4 \\ \hline \end{array}$$

**Remarque 2.56.** *La condition $|x| \leq 1/4$ est impliquée par la contrainte plus forte*

$$|q_i| + |x| \leq 1/4.$$

*Nous pouvons donc l'oublier pour la suite.*

Si on choisit de prendre $d_i^{(0)} = 0$, on a $y_i^{(0)} = 0$ et le système (2.11) nous fournit des conditions initiales.

$$\boxed{\forall i \in [\![0,\, n]\!]\,,\ |p_i| \leq 3/4}$$

*A priori* pour calculer $d_i^{(j)}$, il faut connaître $w_i^{(j)}$ exactement. L'opérateur matériel qui réalise l'arrondi à l'entier le plus proche sera donc assez gros et assez lent. Or la redondance de la représentation des $y_i$ nous autorise en fait à être un peu imprécis. Choisissons un paramètre $\Delta > 0$ et supposons que $\widehat{w_i}^{(j)}$ soit une approximation de $w_i^{(j)}$ avec une erreur absolue inférieure à $\Delta/2$ (le plus simple est de tronquer $w_i^{(j)}$ après quelques chiffres par exemple) et supposons qu'on choisisse $d_i^{(j)}$ en considérant $\widehat{w_i}^{(j)}$ au lieu de $w_i^{(j)}$. On veut assurer par induction que $|w_i^{(j)}|$ reste borné par $3/2 + \Delta/2$. On distingue alors deux cas :

a) Si $|\widehat{w_i}^{(j)}| \leq 3/2$, on choisit $d_i^{(j)}$ égal à l'entier le plus proche de $\widehat{w_i}^{(j)}$ et on a donc

$$\left|w_i^{(j)} - d_i^{(j)}\right| = \left|w_i^{(j)} - \widehat{w_i}^{(j)} + \widehat{w_i}^{(j)} - d_i^{(j)}\right| \leq (1+\Delta)/2.$$

b) Si $|\widehat{w_i}^{(j)}| > 3/2$, puisque

$$|\widehat{w_i}^{(j)}| - \frac{\Delta}{2} \leq |w_i^{(j)}| \leq \frac{3}{2} + \frac{\Delta}{2},$$

on a $|w_i^{(j)}| \in \left[\frac{3}{2} - \frac{\Delta}{2},\, \frac{3}{2} + \frac{\Delta}{2}\right]$. Il suit que

$$\left|w_i^{(j)} - d_i^{(j)}\right| \leq (1+\Delta)/2.$$

Pour garantir que $|w_i^{(j+1)}|$ reste bien borné par $3/2 + \Delta/2$, il faut donc modifier légèrement les conditions que nous avions établies précédemment.

$$\boxed{\begin{aligned} |q_n| &\leq \frac{1-\Delta}{4} \\ \forall i \in [\![1,\, n-1]\!]\,,\ |q_i| + |x| &\leq \frac{1-\Delta}{4} \\ \forall i \in [\![0,\, n]\!]\,,\ |p_i| &\leq \frac{3+\Delta}{4} \end{aligned}}$$

**Remarque 2.57.** *Comme on pouvait s'y attendre, les contraintes sont d'autant plus grandes que $\Delta$ est grand. En particulier, on ne peut pas choisir $\Delta \geq 1$ : on est obligé de choisir le paramètre $\Delta$ dans l'intervalle* $[0,\, 1[$.

L'unité élémentaire (EU) qui calcule $d_i^{(j)}$ et $w_i^{(j+1)}$ est représentée en figure 2.14. Comme on peut le voir, le circuit est extrêmement simple. Les boîtes MG (*multiple generator*) calculent la multiplication d'un nombre par un bit ; il s'agit donc en fait uniquement de multiplexeurs. Le résidu $w_i^{(j)}$ est représenté en notation *carry-save* : $w^{(j)} = w_s^{(j)} + w_c^{(j)}$ ce qui permet d'éviter la propagation de retenue lors de l'addition finale (on pourra consulter [Mul89] pour une présentation de l'addition carry-save). Le bit $d_i^{(j+1)}$ est choisi par un opérateur de sélection qui n'a besoin de considérer que quelques bits de $w_i^{(j+1)}$.

L'architecture générale de calcul des $y_i$ est représentée en figure 2.15. Il s'agit d'un circuit extrêmement régulier où les unités élémentaires EU calculent chacune en parallèle et à chaque cycle d'horloge un bit de $y_i$. Chaque bit qui sort pour $y_0$ est accumulé dans la boîte OFC (*on-the-fly converter*) qui a donc pour sortie la valeur $y_0^{(j)}$ correspondant à un développement tronqué de la valeur exacte $y_0 = R(x)$.

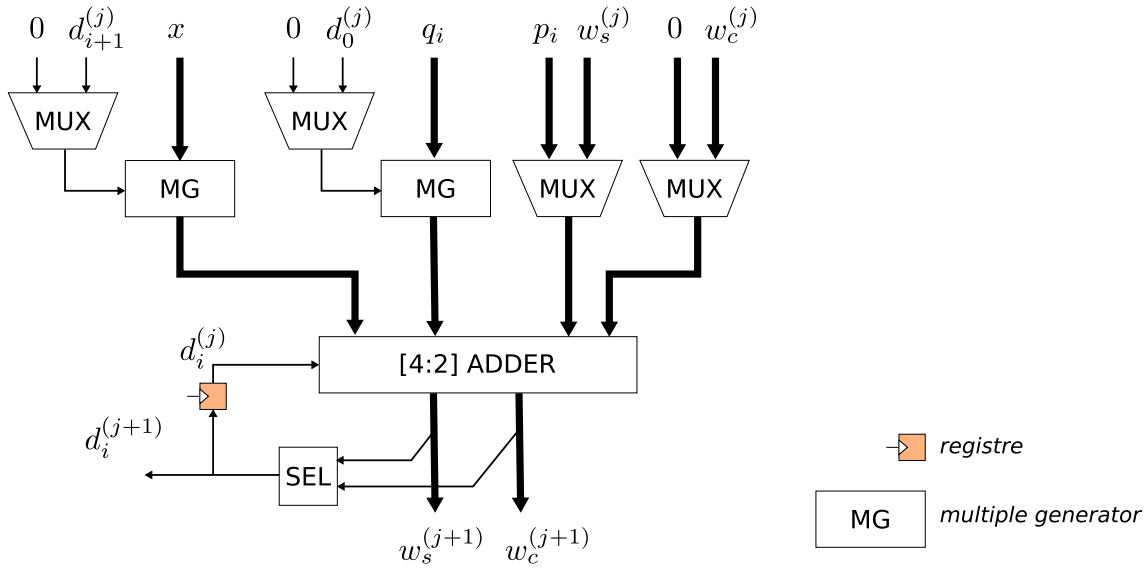



**Figure 2.14 :** Unité élémentaire pour le calcul d'un bit de $y_i$ dans la E-méthode. Les multiplexeurs sont là pour différencier le cas $j = 0$ du cas $j > 0$.

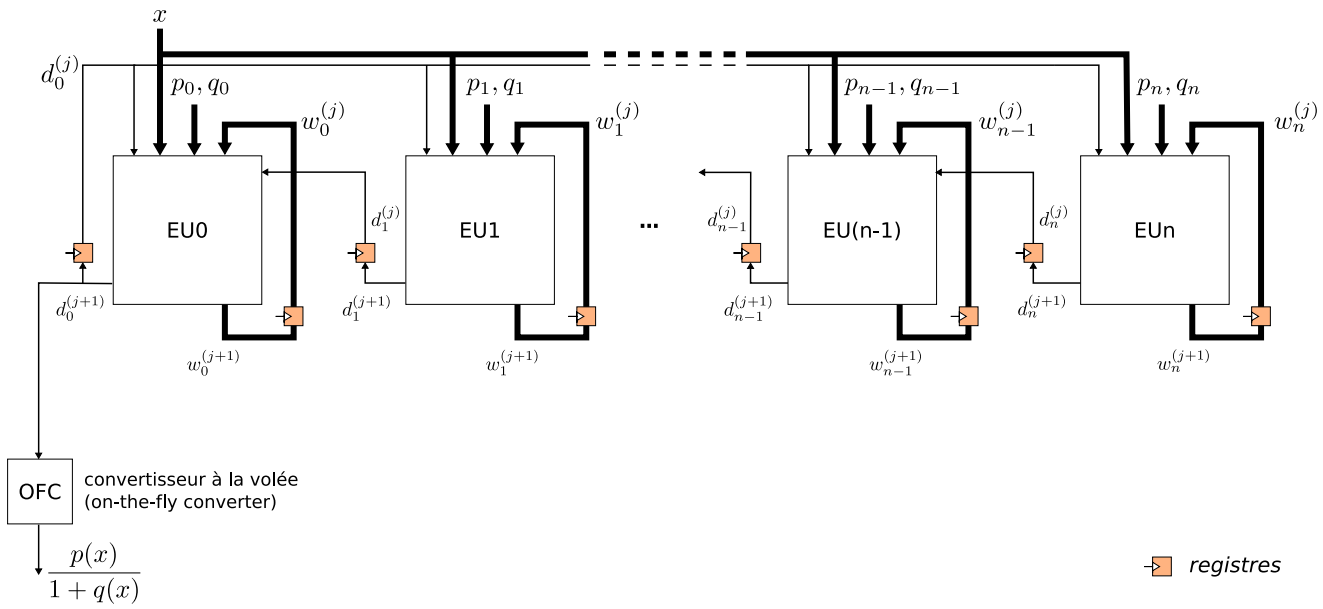



**Figure 2.15 :** Architecture générale d'un circuit implémentant la E-méthode

### 2.4.2 Approximation rationnelle et E-méthode

Comme nous l'avons vu, la E-méthode présente l'avantage d'évaluer la fraction rationnelle $R$ à l'aide d'un circuit simple et régulier qui fournit un bit du résultat à chaque cycle d'horloge. Cependant, il faut que la fraction rationnelle vérifie certaines contraintes. Le développeur peut bien sûr choisir $\Delta$ comme il le souhaite dans $[0,\, 1[$, mais il faut alors s'assurer que la fraction rationnelle à évaluer vérifie bien les contraintes de la E-méthode.

Nous nous intéressons donc à présent au problème de la recherche de fractions rationnelles $R^\star$ approchant le mieux possible une fonction $f$ en norme sup, et respectant, par ailleurs, les contraintes de la E-méthode. Dans un premier temps, nous cherchons à trouver les meilleures approximations à coefficients réels. Dans un second temps, nous ajouterons à cela la contrainte d'avoir des coefficients représentables en machine.

**Contraintes sur le numérateur.** Remarquons d'abord que les contraintes sur les $p_i$ ne constituent pas un réel problème. En effet, il est toujours possible de choisir un entier $e$ de telle sorte que

$$\begin{aligned} R^\star(x) &= \frac{p_0 + p_1 x + \cdots + p_n x^n}{1 + q_1 x + \cdots + q_n x^n} \\ &= 2^e \, \frac{\left(\frac{p_0}{2^e}\right) + \left(\frac{p_1}{2^e}\right) x + \cdots + \left(\frac{p_n}{2^e}\right) x^n}{1 + q_1 x + \cdots + q_n x^n} \\ &= 2^e \, \widetilde{R^\star}(x) \end{aligned}$$

où le numérateur de $\widetilde{R^\star}$ remplit les conditions de la E-méthode. En pratique, on évaluera donc $\widetilde{R^\star}$ par la E-méthode et on fera la multiplication par $2^e$ (qui n'est rien d'autre qu'un décalage de la virgule) pour obtenir le résultat souhaité.

**Contraintes sur le dénominateur.** Les vraies contraintes de la E-méthode portent donc en fait sur le dénominateur de la fraction. Ces contraintes peuvent s'écrire sous la forme $|q_i| \leq \beta_i$, où $\beta_i$ est une constante. En effet, pour $q_n$ c'est déjà le cas ; pour les autres $q_i$, puisque la contrainte $|q_i| + |x| \leq (1 - \Delta)/4$ doit être vérifiée pour tout $x$ dans $[a,\, b]$, elle est équivalente à

$$|q_i| \leq \beta_i = \frac{1 - \Delta}{4} - \max\{|a|,\, |b|\}.$$

Nous cherchons donc un algorithme qui calcule une fraction de meilleure approximation (en norme sup) de la forme

$$R^\star(x) = \frac{p_0 + \cdots + p_n x^n}{1 + q_1 x + \cdots + q_n x^n} = \frac{p(x)}{1 + q(x)}$$

sous la contrainte $|q_i| \leq \beta_i$. Ce problème est assez général et ne semble pas très éloigné du problème classique de recherche de fractions rationnelles de meilleure approximation, pour lequel il existe des algorithmes de type *algorithme de Rémez* (cf. [Che82, Chap. 5] ou [Pow81, Chap. 10]). Il semble donc probable qu'un tel algorithme existe pour résoudre ce problème, mais nous n'en avons pas connaissance et n'avons pas eu le temps de le mettre en évidence.

Nous utilisons donc pour l'instant un algorithme imparfait, et pour tout dire assez inefficace, qui repose sur la résolution successive de problèmes de programmation linéaire. Notre algorithme ne résout pas le problème continu mais le problème discret : étant donnés $k$ points $x_1, \ldots, x_k$, trouver une fraction rationnelle $R$ de la forme désirée qui minimise

$$\max_{i \in [\![1,\, k]\!]} |R(x_i) - f(x_i)|.$$

Nous supposerons que le problème continu (minimisation de $\|R - f\|_\infty$) et le problème discret se confondent en pratique si les points $x_i$ sont suffisamment nombreux et bien répartis dans $[a, b]$. Dans la suite, on notera $\varepsilon^\star$ l'erreur optimale du problème discret.

Pour résoudre le problème discret, considérons un réel positif $\varepsilon$. Nous cherchons à déterminer si $\varepsilon \geq \varepsilon^\star$ ou $\varepsilon < \varepsilon^\star$. Pour cela, nous commençons par montrer le lemme suivant :

**Lemme 2.58.** *Si $p/(1+q)$ satisfait aux contraintes de la E-méthode sur $[a, b]$, on a*

$$\forall x \in [a, b]\,,\ |q(x)| \leq (1 - \Delta)/4.$$

*Démonstration.* Considérons la suite $(v_i)$ correspondant à l'évaluation de $q(x)$ par le schéma de Horner : $v_n = q_n$ et pour $i \geq 0$, $v_i = q_i + x\, v_{i+1}$. Par construction, $q(x) = x\, v_1$.

Nous montrons par récurrence que tous les $v_i$ vérifient $|v_i| \leq (1 - \Delta)/4$. C'est vrai de $v_n = q_n$ (c'est une des contraintes de la E-méthode). Soit donc $i \geq 1$. On suppose que $|v_{i+1}| \leq (1 - \Delta)/4$. Cela implique que $|v_{i+1}| \leq 1$. On a alors

$$|v_i| = |q_i + x\, v_{i+1}| \leq |q_i| + |x| \cdot |v_{i+1}| \leq |q_i| + |x|.$$

Les contraintes de la E-méthode impliquent que $|q_i| + |x| \leq (1 - \Delta)/4$, ce qui termine la récurrence.

On en déduit que $|v_1| \leq (1 - \Delta)/4$ et donc $|q(x)| = |x\, v_1| \leq |x|\,(1 - \Delta)/4$. On conclut en remarquant que $|x| \leq 1/4$ (contrainte de la E-méthode). □

On déduit de ce lemme que $\forall x \in [a, b]\,,\ 1 + q(x) > 0$. Il suit que, pour $x \in [a, b]$,

$$\left|\frac{p(x)}{1+q(x)} - f(x)\right| \leq \varepsilon$$

$$\iff \begin{cases} p(x)/(1+q(x)) - f(x) & \leq \varepsilon \\ f(x) - p(x)/(1+q(x)) & \leq \varepsilon \end{cases}$$

$$\iff \begin{cases} p(x) - f(x) - f(x)q(x) & \leq \varepsilon(1+q(x)) \\ f(x) + f(x)q(x) - p(x) & \leq \varepsilon(1+q(x)). \end{cases}$$

Par suite, $\varepsilon \geq \varepsilon^\star$ si et seulement s'il existe deux polynômes $p$ et $q$ tels que

$$\begin{cases} -\beta_i \leq q_i \leq \beta_i, & i \in [\![1, n]\!] \\ p(x_i) - f(x_i) - f(x_i)q(x_i) \leq \varepsilon + \varepsilon q(x_i), & i \in [\![1, k]\!] \\ f(x_i) + f(x_i)q(x_i) - p(x_i) \leq \varepsilon + \varepsilon q(x_i), & i \in [\![1, k]\!]\,. \end{cases} \tag{2.13}$$

Ce système est constitué d'un nombre fini de contraintes linéaires sur les coefficients de $p$ et $q$ : il s'agit donc juste de déterminer s'il existe un point faisable dans un problème de programmation linéaire. Nous pouvons le tester en utilisant n'importe quel programme de programmation linéaire, comme l'algorithme du simplexe par exemple [Sch86].

Considérons d'abord l'erreur $\varepsilon_\ell$ fournie par la meilleure approximation rationnelle de $f$ sans contrainte. Nous pouvons la calculer en utilisant une variante de l'algorithme de Rémez pour les fractions rationnelles (voir [Che82, Chap. 5] ou [Pow81, Chap. 10] par exemple). Par définition, $\varepsilon_\ell \leq \varepsilon^\star$.

Nous commençons donc par poser $\varepsilon = 2\varepsilon_\ell$ et testons la faisabilité du problème de programmation linéaire associé. S'il n'est pas faisable, nous posons $\varepsilon = 4\varepsilon_\ell$, etc. jusqu'à trouver une valeur $\varepsilon_u$ pour laquelle le problème de programmation linéaire admet un point faisable.

À partir de là, nous disposons à la fois d'un minorant $\varepsilon_\ell$ et d'un majorant $\varepsilon_u$ de la valeur optimale $\varepsilon^\star$. Nous pouvons donc nous approcher de $\varepsilon^\star$ d'aussi près que nous voulons en procédant par dichotomie :

— si le problème associé à $\varepsilon = (\varepsilon_\ell + \varepsilon_u)/2$ a un point faisable, on pose $\varepsilon_u = (\varepsilon_\ell + \varepsilon_u)/2$ ;
— s'il n'en n'a pas, on pose $\varepsilon_\ell = (\varepsilon_\ell + \varepsilon_u)/2$, etc.

Une fois obtenue une valeur $\varepsilon_u$ suffisamment proche de $\varepsilon^\star$, n'importe quel point faisable du problème associé à $\varepsilon_u$ fournit des coefficients $p_i$ et $q_i$ vérifiant les contraintes de la E-méthode et pour lesquels

$$\max_{i\in[\![1,\,k]\!]} |R^\star(x_i) - f(x_i)| \leq \varepsilon_u.$$

Puisque $\varepsilon_u \simeq \varepsilon^\star$ et $\max_i |R^\star(x_i) - f(x_i)| \simeq \|R^\star - f\|_\infty$, on a trouvé une fraction rationnelle vérifiant les contraintes de la E-méthode et ayant une qualité d'approximation quasi optimale.

### 2.4.3 Approximation rationnelle à coefficients machine

Nous nous intéressons à présent au problème de la représentabilité des coefficients de la fraction rationnelle en machine. Pour ce faire, nous adaptons la méthode à base de réseaux euclidiens présentée en section 2.3.4 page 113.

Cette méthode consistait à interpoler de manière approchée le « polynôme » de meilleure approximation $p^\star$ par un « polynôme » à coefficients entiers, la notion de « polynôme » étant prise au sens généralisé de combinaison linéaire de $n+1$ fonctions de base $\varphi_0, \ldots, \varphi_n$.

Dans le cas présent, la transposition naturelle consiste donc à essayer d'interpoler la fraction rationnelle $R^\star$ de la section précédente par une fraction rationnelle dont les coefficients sont des nombres machine. Ce faisant, nous espérons que la fraction rationnelle $R$ obtenue ressemblera à $R^\star$ (et donc approchera la fonction $f$ de façon quasi optimale) et continuera de respecter les contraintes de la E-méthode.

Si les coefficients de la fraction rationnelle doivent être des nombres flottants, on peut deviner les exposants des coefficients comme nous l'avions fait dans le cas polynomial. De la même façon qu'avec les polynômes, on peut aussi les déterminer rigoureusement en utilisant un polytope (celui de l'équation (2.13)) et en projetant orthogonalement sur chaque axe.

Cependant, la E-méthode étant destinée à une évaluation en matériel et en virgule fixe, on s'attend plutôt à ce que les coefficients soient eux-mêmes en virgule fixe et donc que l'exposant ait été choisi par l'utilisateur *a priori*. On peut donc supposer que les exposants sont connus et se ramener ainsi au problème entier suivant : trouver des entiers relatifs $a_0, \ldots, a_n, b_1, \ldots, b_n$ tels que la fraction

$$R(x) = \frac{a_0 2^{e_0} + \cdots + a_n(2^{e_n} x^n)}{1 + b_1(2^{f_1} x) + \cdots + b_n(2^{f_n} x^n)}$$

soit une interpolation approchée de $R^\star$.

De la même façon que pour la méthode décrite en section 2.3.4, nous choisissons $2n+1$ points $z_0, \ldots, z_{2n}$ dans l'intervalle $[a, b]$ (on peut éventuellement en choisir plus) et nous écrivons la propriété désirée, à savoir

$$\forall j \in [\![0,\, 2n]\!]\,,\ \frac{a_0 2^{e_0} + \cdots + a_n(2^{e_n} z_j^n)}{1 + b_1(2^{f_1} z_j) + \cdots + b_n(2^{f_n} z_j^n)} \simeq R^\star(z_j).$$

De façon à nous ramener à un système linéaire, nous multiplions les deux membres par le dénominateur et obtenons le système linéaire suivant :

$$a_0 \begin{pmatrix} 2^{e_0} \\ \vdots \\ 2^{e_0} \end{pmatrix} + \cdots + a_n \begin{pmatrix} 2^{e_n} z_0^n \\ \vdots \\ 2^{e_n} z_{2n}^n \end{pmatrix} - b_1 \begin{pmatrix} 2^{f_1} z_0 R^\star(z_0) \\ \vdots \\ 2^{f_1} z_{2n} R^\star(z_{2n}) \end{pmatrix} - \cdots - b_n \begin{pmatrix} 2^{f_n} z_0^n R^\star(z_0) \\ \vdots \\ 2^{f_n} z_{2n}^n R^\star(z_{2n}) \end{pmatrix} \simeq \begin{pmatrix} R^\star(z_0) \\ \vdots \\ R^\star(z_{2n}) \end{pmatrix} \tag{2.14}$$

Nous sommes à nouveau ramenés à un problème de recherche d'un plus proche vecteur dans un réseau, que nous résolvons de manière approchée à l'aide de l'algorithme LLL comme nous l'avions fait en section 2.3.4.

Comme dans le cas des polynômes, le choix des points $z_0, \ldots, z_{2n}$ est libre, mais *a priori* le choix le plus malin semble être, lorsque c'est possible, de choisir des points $z_j$ tels que $R^\star(z_j) = f(z_j)$. À défaut, les points de Tchebychev constituent toujours un choix privilégié.

### 2.4.4 Résultats expérimentaux

Nous avons implémenté les algorithmes décrits dans les deux sections précédentes sous forme de prototypes. Notre implémentation n'est pas du tout automatisée et repose encore sur l'utilisation de nombreux scripts et une intervention manuelle. Pour rechercher la meilleure fraction rationnelle à coefficients réels sans contrainte, nous utilisons la procédure `minimax` du module `numapprox` de Maple. Pour rechercher la fraction $R^\star$ à coefficients réels vérifiant les contraintes de la E-méthode (section 2.4.2), nous utilisons la méthode dichotomique que nous avons décrite. Là encore, Maple est utilisé pour tout ce qui concerne la partie programmation linéaire. Pour trouver une fraction rationnelle à coefficients représentables sous forme de nombres machine (section 2.4.3), nous utilisons un script qui s'appuie sur l'outil GP [10] et son implémentation de l'algorithme LLL.

Les résultats expérimentaux sont moins probants qu'ils ne l'étaient dans le cas polynomial. Étudions deux exemples ; dans chacun, on recherche des fractions rationnelles dont les coefficients sont stockés en virgule fixe sous la forme $a/2^{24}$, où $a$ est un entier.

**Exemple 1.** Il s'agit d'approcher la fonction sinh sur l'intervalle $[0, 1/8]$ par une fraction rationnelle dont le numérateur est de degré 3 et le dénominateur est de degré 4. La valeur $\Delta$ est prise égale à $1/2$. La fraction rationnelle renvoyée par la procédure `minimax` de Maple fournit une erreur absolue de $6.35\,\mathrm{e}{-18}$. Elle vérifie déjà les conditions de la E-méthode et il n'est donc pas nécessaire d'utiliser la technique dichotomique sur cet exemple.

La fraction rationnelle ainsi obtenue est $R^\star = (p_0+p_1x+p_2x^2+p_3x^3)/(1+q_1x+q_2x^2+q_3x^3+q_4x^4)$ avec

$$\begin{aligned}
p_0 &\simeq -6.35249895080230104246781 16\,\mathrm{e}{-18}, & q_1 &\simeq -1.30077963217001986014934 07\,\mathrm{e}{-5},\\
p_1 &\simeq 1.00000000000000063154033793, & q_2 &\simeq -6.11767712515252644205584 51\,\mathrm{e}{-2},\\
p_2 &\simeq -1.30077973501032583601649 08\,\mathrm{e}{-5}, & q_3 &\simeq 2.16986784442664688787663 73\,\mathrm{e}{-6},\\
p_3 &\simeq 0.10548989547859930926778 73, & q_4 &\simeq 1.86276494542991328904228 92\,\mathrm{e}{-3}.
\end{aligned}$$

Si on arrondit chaque coefficient au multiple entier de $2^{-24}$ le plus proche, on obtient une fraction rationnelle $\widehat{R}$ qui fournit une erreur de $4.41\,\mathrm{e}{-11}$, ce qui représente une perte de précision d'approximativement 23 bits.

En appliquant l'algorithme qui utilise les réseaux euclidiens et l'algorithme LLL, on obtient la fraction rationnelle dont les coefficients sont les suivants :

$$\begin{aligned}
p_0 &= 0, & q_1 &= -99728/2^{24},\\
p_1 &= 1, & q_2 &= -762113/2^{24},\\
p_2 &= -99728/2^{24}, & q_3 &= 16637/2^{24},\\
p_3 &= 2034090/2^{24}, & q_4 &= -13057/2^{24}.
\end{aligned}$$

Cette fraction fournit une erreur de $9.01\,\mathrm{e}{-14}$. Cela représente un gain de précision d'approximativement 9 bits par rapport à $\widehat{R}$.

10. GP est un outil distribué sous licence GPL et disponible à l'adresse http://pari.math.u-bordeaux.fr/

Comme on peut le voir, l'ordre de grandeur des coefficients de la fraction rationnelle obtenue n'a rien à voir avec l'ordre de grandeur des coefficients correspondant dans $R^\star$. Parfois, le coefficient a même changé de signe : c'est le cas de $q_4$. Il s'agit là d'une différence pratique majeure par rapport au cas polynomial. En effet, dans le cas polynomial, les coefficients finalement obtenus sont, le plus souvent, assez proches des coefficients du polynôme de meilleure approximation.

Ce changement d'ordre de grandeur des coefficients n'est pas de bon augure. En effet, les contraintes de la E-méthode imposent que les coefficients du dénominateur soient bornés par une certaine constante. En partant de la fraction $R^\star$ qui vérifie les contraintes de la E-méthode, on espérait obtenir une fraction dont les coefficients soient représentables en machine, mais restent du même ordre de grandeur que ceux de $R^\star$, et donc vérifient encore les contraintes de la E-méthode.

Dans le cas présent, la fraction obtenue vérifie encore les contraintes de la E-méthode mais c'est, pour ainsi dire, dû au hasard puisque les coefficients n'ont plus rien à voir avec ceux de $R^\star$. De fait, nous allons voir, avec l'exemple suivant, un cas pour lequel la méthode à base de réseaux euclidiens renvoie une fraction qui ne répond plus aux contraintes de la E-méthode.

**Exemple 2.** Cette fois nous voulons approcher arctan sur l'intervalle $[0, \tan(\pi/32)]$ par une fraction rationnelle de la forme $R = (p_0x + p_1x^3 + p_2x^5 + p_3x^7)/(1 + q_1x^2 + q_2x^4 + q_3x^6 + q_4x^8)$. Le paramètre $\Delta$ est fixé à $1/4$. Pour trouver la meilleure approximation rationnelle à coefficients réels sans contraintes, nous faisons appel à Maple. Comme Maple ne permet pas de rechercher une fraction directement sous la forme demandée, on utilise le fait que $R(x)$ se récrit $x\widetilde{R}(x^2)$, où $\widetilde{R}$ a un numérateur de degré 3 et un dénominateur de degré 4. Donc en faisant le changement de variable $y = x^2$ et en utilisant la fonction de poids $x = \sqrt{y}$, on se ramène au cas classique que Maple sait traiter.

On obtient ainsi la fraction rationnelle suivante qui fournit une erreur de $2.63\,\mathrm{e}{-27}$ :

$$p_0 \simeq 0.99999999999999999999999544541, \qquad q_1 \simeq 1.86407908261308822662821444690399 5,$$
$$p_1 \simeq 1.53074574927975489329714488047 8, \qquad q_2 \simeq 1.07339733237310056432257901166274 8,$$
$$p_2 \simeq 0.65203763816873718831804951232, \qquad q_3 \simeq 0.19463040174394329896829947047110 1,$$
$$p_3 \simeq 6.67899646183867018139151372 8\,\mathrm{e}{-2}, \qquad q_4 \simeq 5.38323479819095098294237281865 8\,\mathrm{e}{-3}.$$

Cependant, cette fraction ne satisfait pas aux contraintes de la E-méthode. Nous utilisons donc l'algorithme dichotomique à base de programmation linéaire et obtenons finalement une fraction $R^\star$, dont les coefficients (réels) vérifient les contraintes de la E-méthode, et qui est quasi-optimale en terme d'approximation. Ces coefficients sont les suivants :

$$p_0 \simeq 0.99999999998894880197740569 23, \qquad q_1 \simeq -5.42501729403121182138184428256065 3\,\mathrm{e}{-7},$$
$$p_1 \simeq -0.33333386058048754594909764 90, \qquad q_2 \simeq -4.66514613373431110324005469767409 4\,\mathrm{e}{-4},$$
$$p_2 \simeq 0.19952797296869948328220836 87, \qquad q_3 \simeq -3.68250851259001356640907690464674 9\,\mathrm{e}{-2},$$
$$p_3 \simeq -0.17873640396485984535423804 30, \qquad q_4 \simeq -8.60417830530885797592377683641848\,\mathrm{e}{-2}.$$

Cette fraction fournit une erreur d'approximation de $1.207\,\mathrm{e}{-13}$, ce qui est beaucoup moins bien ; cependant, par construction, c'est ce qu'on peut espérer de mieux si on veut vérifier les contraintes de la E-méthode. La fraction obtenue en remplaçant chaque coefficient de $R^\star$ par le multiple entier de $2^{-24}$ le plus proche fournit une erreur de $5.55\,\mathrm{e}{-12}$.

Nous essayons de faire mieux en utilisant l'algorithme à base de réseaux euclidiens avec la fraction $R^\star$ et en choisissant pour points d'interpolation les points de Tchebychev. L'algorithme renvoie une fraction dont l'erreur d'approximation est $1.207\,\mathrm{e}{-13}$, c'est-à-dire une fraction dont la qualité d'approximation est la même que $R^\star$ (à très peu de chose près) mais avec des coefficients représentables. La courbe d'erreur de cette fraction est la même en pratique que la courbe d'erreur de $R^\star$. L'interpolation approchée a donc parfaitement bien fonctionné.

Malheureusement, les coefficients de cette fraction sont :

$$\begin{aligned} p_0 &\simeq 1, & q_1 &\simeq 171780615429748345/2^{24},\\ p_1 &\simeq 171780615422257555/2^{24}, & q_2 &\simeq 170601323782957478/2^{24},\\ p_2 &\simeq 113341121261628151/2^{24}, & q_3 &\simeq 93509861616815062/2^{24},\\ p_3 &\simeq 70997901374518732/2^{24}, & q_4 &\simeq 21454962956816228/2^{24}. \end{aligned}$$

Ces coefficients sont immenses (l'ordre de grandeur de $q_1$ est $10^{10}$) et ils ne remplissent donc pas du tout les conditions de la E-méthode. L'algorithme LLL a donc trouvé avec succès une fraction rationnelle (de bonne qualité d'approximation, résolvant correctement le problème d'interpolation approchée, et dont les coefficients sont bien des multiples entiers de $2^{24}$) mais les coefficients de cette fraction n'ont plus rien à voir avec ceux de la fraction $R^\star$ qu'on cherchait à imiter.

On ne peut donc pas l'utiliser dans le cadre de la E-méthode et on doit se contenter de la fraction $\widehat{R}$ dont l'erreur est $5.55\,\mathrm{e}{-12}$.

Pour résoudre ce problème, il faudrait trouver un moyen de contraindre l'algorithme LLL à rester dans le voisinage de $R^\star$. Nous n'avons pas trouvé pour l'instant de méthode convenable pour forcer l'algorithme à avoir un tel comportement.

## 2.5 Conclusion de ce chapitre

Nous avons étudié plusieurs problèmes en rapport avec l'approximation polynomiale (ou rationnelle) utilisée pour l'évaluation de fonctions en précision fixée.

Lorsqu'on souhaite écrire un programme pour évaluer une fonction $f$ sur un intervalle $[a, b]$, on commence généralement par la remplacer par un polynôme $p$. Étant donné un degré maximal $n$, le problème naturel consiste alors à rechercher le polynôme $p^\star$ de degré inférieur ou égal à $n$ pour lequel l'erreur relative $\|p/f - 1\|_\infty$ est minimale (ou $\|p - f\|_\infty$ dans le cas, moins fréquent, de l'erreur absolue). Un algorithme itératif dû à Rémez permet de calculer $p^\star$.

Les développeurs de bibliothèques mathématiques expriment souvent des besoins particuliers : par exemple, ils peuvent souhaiter fixer à l'avance la valeur de certains coefficients. Le problème est alors légèrement modifié : au lieu de chercher $p^\star$ comme une combinaison linéaire des monômes $\varphi_j(x) = x^j$, on le cherche comme combinaison de *certains* monômes : $\varphi_j(x) = x^{\alpha_j}$. Ceci nous a amené à considérer le problème général suivant : étant données $n+1$ fonctions continues quelconques $\varphi_0, \ldots, \varphi_n$ et une fonction continue à approcher $g$, trouver $p^\star = \sum_{j=0}^{n} a_j\,\varphi_j$ pour lequel la norme $\|p^\star - g\|_\infty$ est minimale.

Si le système $(\varphi_0, \ldots, \varphi_n)$ vérifie une certaine condition appelée *condition de Haar*, l'algorithme de Rémez fonctionne encore. Lorsque la condition de Haar n'est pas vérifiée, il faut utiliser des généralisations dues à Stiefel, Descloux, Laurent et Carasso, ou Watson et Osborne. Nous avons implémenté une telle généralisation dans le logiciel Sollya. Bien que nous n'ayons pas de garantie sur la convergence de l'algorithme, on observe qu'il fonctionne bien en pratique. Notre implémentation est raisonnablement efficace et a été utilisée avec succès pour développer une partie de la bibliothèque `CRlibm`. Il ne faut pas plus de quelques secondes, en général, pour que l'algorithme fournisse un polynôme de meilleure approximation sur les problèmes issus de `CRlibm` (pour lesquels, typiquement, $n$ est de l'ordre de 20 et $[a, b] = [-1/16, 1/16]$). Nous envisageons une récriture complète de notre implémentation qui devrait la rendre très efficace.

En pratique, on ne peut stocker et évaluer que les polynômes dont les coefficients sont des nombres machine. Une idée consiste à calculer $p^\star$, puis arrondir chacun de ses coefficients au nombre machine le plus proche. Cependant, le polynôme $\widehat{p}$ ainsi obtenu peut être grandement sous-optimal. Nous avons proposé deux approches complémentaires pour trouver de très bons (voire les meilleurs) polynômes d'approximation à coefficients machine. La première approche s'appuie sur des idées de

programmation linéaire pour fournir des résultats rigoureux. Ainsi, il est possible de minorer rigoureusement et assez finement l'erreur $\varepsilon_{\text{opt}}$ optimale (parmi les polynômes à coefficients machine) : $\alpha \leq \varepsilon_{\text{opt}}$. Si l'on connaît déjà un polynôme $p$ qui vérifie les contraintes de format et que, par exemple, $\|p - g\|_\infty = 1,1 \cdot \alpha$, on en déduit que

$$\varepsilon_{\text{opt}} \leq \|p - g\|_\infty \leq 1,1 \cdot \varepsilon_{\text{opt}}.$$

Cet encadrement est obtenu sans qu'on connaisse explicitement la valeur de $\varepsilon_{\text{opt}}$. On peut alors éventuellement se satisfaire de $p$ puisqu'on sait qu'il fournit une erreur à moins de 10% de l'optimum. Parfois, il est également possible, en couplant cette méthode avec des recherches exhaustives, de déterminer la liste de tous les polynômes optimaux.

La deuxième approche repose sur l'utilisation de l'algorithme LLL pour obtenir un très bon polynôme $p$ à coefficients machine. Notre méthode est heuristique car, *a priori*, il n'y a aucune garantie que $p$ approchera correctement $g$. Cependant, en pratique elle donne d'excellents résultats. Cet algorithme a été implanté sous la forme de la commande `fpminimax` dans Sollya. Sur les problèmes rencontrés dans le développement de `CRlibm`, il faut environ une seconde à l'algorithme pour calculer un polynôme quasi-optimal.

Les deux approches proposées peuvent être utilisées conjointement pour fournir rapidement des résultats certifiés : ainsi, `fpminimax` peut fournir rapidement un polynôme $p$ ; les outils de programmation linéaire peuvent alors confirmer rigoureusement que $p$ fournit une erreur proche de l'optimum. Le cas échéant, il est même possible d'obtenir des informations du type « la valeur du premier coefficient est forcément tel nombre flottant ». Dans ce cas, `fpminimax` peut être utilisé à nouveau, en prenant en compte cette contrainte additionnelle pour fournir un meilleur polynôme, et ainsi de suite.

Dans la dernière partie du chapitre, nous nous sommes intéressés à l'évaluation efficace de fractions rationnelles en matériel. Pour cela, nous utilisons la E-méthode, proposée par Ercegovac. Pour que cette méthode fonctionne, il faut que les coefficients de la fraction rationnelle vérifient certaines contraintes. Nous avons proposé un algorithme pour calculer la meilleure fraction d'approximation, parmi celles vérifiant les contraintes de la E-méthode. En outre, nous avons adapté l'algorithme `fpminimax` pour pouvoir chercher des fractions d'approximations à coefficients machine.

Les algorithmes présentés dans ce chapitre ont prouvé leur intérêt pratique ; cependant, ils sont encore perfectibles sur bien des plans. L'algorithme `fpminimax` est heuristique, et sur quelques rares cas, nous avons observé qu'il bouclait indéfiniment ou qu'il renvoyait un polynôme de piètre qualité. Il faudra donc comprendre ces phénomènes et tenter d'y remédier.

Il serait intéressant de trouver une caractérisation des polynômes optimaux (ou presque optimaux) à coefficients machine. De la même façon que l'algorithme de Rémez s'appuie sur le théorème d'alternance et le théorème de La Vallée Poussin, nous pourrions peut-être nous appuyer sur une caractérisation pour rechercher les bons polynômes dont les coefficients vérifient les contraintes de format.

L'algorithme `fpminimax` permet de trouver rapidement de très bons polynômes : autrement dit, il est relativement aisé de trouver un majorant fin de l'erreur optimale $\varepsilon_{\text{opt}}$. En revanche, il est plus délicat de calculer un minorant fin de $\varepsilon_{\text{opt}}$. Les outils de programmation linéaire nous permettent de calculer de tels minorants, mais la méthode est encore très manuelle et manque de systématisme. Une méthode pour calculer automatiquement et rapidement ce genre de minorants serait particulièrement appréciable.

Pour finir, nos travaux sur l'approximation rationnelle et la E-méthode n'en sont encore qu'à un stade très préliminaire. Nous avons deux algorithmes indépendants : le premier calcule une fraction $R^\star$ à coefficients réels et quasi-optimale parmi les fractions satisfaisant les contraintes de la E-méthode. Le second est une adaptation de l'algorithme `fpminimax` pour trouver des fractions rationnelles à coefficients machine.

Pour l'instant, nous n'avons pas trouvé de moyen pour forcer l'algorithme `fpminimax` à renvoyer une fraction rationnelle vérifiant les contraintes de la E-méthode. Pour avoir un algorithme réellement satisfaisant, il faudra trouver un moyen de forcer l'algorithme LLL à rester dans le voisinage de la fraction $R^\star$ qu'on lui donne en entrée.

Le calcul de la fraction $R^\star$ lui-même n'est pas pleinement satisfaisant : il est lourd à effectuer car il nécessite de choisir un grand nombre de points d'échantillonnage et de résoudre successivement plusieurs problèmes de programmation linéaire. En outre, il ne renvoie pas une fraction de meilleure approximation sur $[a, b]$ tout entier, mais uniquement sur le sous-ensemble de points considérés. Il y a lieu de penser qu'on peut concevoir un algorithme d'échange, de type *algorithme de Rémez*, où l'on n'aurait besoin que d'un faible nombre de points d'échantillonnage qu'on modifierait à chaque étape de l'algorithme, de façon à véritablement converger vers une fraction de meilleure approximation sur l'intervalle tout entier.

Chapitre 3

# Vers des implantations sûres : le calcul certifié de la norme sup de la fonction d'erreur

*Nous avons vu dans les chapitres précédents comment obtenir une très bonne approximation polynomiale $p$ d'une fonction $f$. Si l'on veut garantir la qualité de l'implantation finale, il faut, entre autres, garantir que l'erreur (absolue ou relative) $\varepsilon(x)$ entre $p(x)$ et $f(x)$ n'est jamais plus grande qu'un certain seuil, lorsque $x$ varie dans un intervalle $[a, b]$. Ce problème se réduit à celui du calcul d'un majorant fin de la norme sup de la fonction d'erreur $\varepsilon$ sur $[a, b]$. Dans ce chapitre, nous nous intéressons donc à la question du calcul certifié de cette norme.*
*Nous rappelons d'abord les algorithmes classiques qui permettent de calculer un encadrement du maximum global d'une fonction sur un intervalle. Nous expliquons en quoi la nature particulière de la fonction $\varepsilon$ (obtenue par soustraction de deux fonctions très proches l'une de l'autre) rend ces algorithmes inutilisables dans le cas qui nous intéresse. Enfin, nous proposons un algorithme pour calculer automatiquement et efficacement un encadrement de la norme sup de ce genre de fonctions sur un intervalle. Notre algorithme utilise des techniques apparentées à la différentiation automatique ainsi que des techniques permettant d'isoler avec certitude les racines d'un polynôme comprises dans un intervalle.*

## 3.1 Situation générale du problème

Jusqu'à présent, nous nous sommes essentiellement intéressés à la synthèse de fonctions d'approximation. Pour cela nous avons eu recours à un important arsenal théorique (théorie de l'approximation polynomiale avec ou sans condition de Haar, programmation linéaire, théorie des réseaux euclidiens). La mise en application de tous ces algorithmes représente plusieurs milliers de lignes de codes, l'usage d'heuristiques, d'algorithmes numériques de résolution d'équations, etc. Nous ne pouvons en aucun cas faire une confiance aveugle à tous ces outils théoriques et pratiques. Nous devons donc nous assurer *a posteriori* que l'approximation proposée répond bien aux contraintes fixées :

— contraintes structurelles : il s'agit de vérifier que certains coefficients ont bien la valeur qui leur avait été prescrite, que la fonction d'approximation est bien écrite dans la base que l'on désirait (cas d'une base « à trous » par exemple). Ces contraintes sont rapidement et facilement vérifiées ;

— contraintes discrètes : il s'agit simplement de vérifier que chaque coefficient est bien représentable dans le format demandé (nombre flottant de précision donnée, expansion, virgule

fixe, absence de dépassement de capacité). Là encore, un examen de chaque coefficient l'un après l'autre permet de vérifier facilement que les contraintes sont respectées ;

— qualité d'approximation suffisante : l'approximation $p$, obtenue par une suite d'algorithmes et d'heuristiques complexes, sert, *in fine*, à approcher une fonction $f$. On doit donc s'assurer que $p$ fournit une qualité d'approximation suffisante. C'est de ce sujet que nous allons discuter dans ce chapitre.

Comme nous l'avons vu avec l'exemple des fonctions erf et erfc, deux sources d'erreurs doivent être considérées : l'erreur d'approximation (entre $p$ et $f$) et l'erreur d'évaluation (erreurs dues aux arrondis commis lors de l'évaluation de la fonction d'approximation). Lorsqu'on veut garantir la qualité finale d'une implémentation, il faut prouver que la somme de ces deux erreurs ne dépasse pas un certain seuil jugé critique. Par exemple, dans le cas où l'on veut promettre à l'utilisateur l'arrondi correct, une analyse préalable des pires cas pour l'arrondi permet de définir une borne d'erreur $\varepsilon$ ; si $y$ est une valeur approchée de la valeur exacte $f(x)$ avec une erreur inférieure à $\varepsilon$, l'arrondi de $y$ à la précision cible est égal à l'arrondi de $f(x)$ (cf. page 78).

Pour offrir toutes les garanties nécessaires, il faut donc trouver un majorant $\varepsilon_1$ de l'erreur d'approximation et un majorant $\varepsilon_2$ de l'erreur d'évaluation. Il est important de remarquer qu'une simple estimation numérique de ces erreurs n'est pas satisfaisante : lorsqu'on affirme fournir l'arrondi correct, on affirme qu'une certaine propriété mathématique est vérifiée par l'implémentation. Et comme pour toute propriété mathématique, une simple affirmation ou un calcul numérique ne suffit pas : il faut une preuve.

Il faut majorer les erreurs ; cependant, cette majoration doit être la plus fine possible. En effet, une majoration large, si elle ne nuit pas à la sécurité du code, nuit en revanche à ses performances. Si on surestime l'erreur totale, on peut se croire obligé d'utiliser une plus grande précision dans les calculs ou un polynôme d'approximation de degré plus élevé, alors que ce n'est pas réellement nécessaire.

Idéalement, on veut donc un encadrement de l'erreur : imaginons qu'on calcule un minorant $\ell$ et un majorant $u$ de l'erreur. Alors, $u$ peut être vu comme une valeur approchée de l'erreur dont la qualité est $(u-\ell)/\ell$. On retrouve ici le même type de raisonnement que nous avons déjà rencontré avec le théorème de La Vallée Poussin (cf. discussion autour de l'équation (2.2), page 84). Si $u$ est une bonne estimation de l'erreur, on sait qu'on ne dépensera pas du temps de calcul ou de la surface de silicium en vain.

Pour ce qui est de la question de l'erreur d'arrondi, les travaux de thèse de Guillaume Melquiond [Mel06] ont considérablement simplifié la tâche. Auparavant, il fallait faire une analyse d'erreur rigoureuse à la main, dans le même genre que celle que nous avons faite pour les fonctions erf et erfc. Cependant, pour erf et erfc, la précision utilisée était la même pendant tout le calcul de la somme. À l'inverse, pour une implémentation de fonction en précision `double` (comme dans `CRlibm`) ou en matériel, on essaie de choisir la précision au plus juste et elle peut donc changer à chaque opération. L'analyse d'erreur doit par conséquent être faite méticuleusement en tenant compte des précisions intermédiaires. Ce travail est très laborieux et peut se révéler source de nombreuses erreurs.

Melquiond a proposé un outil appelé Gappa [1] qui permet d'automatiser ce processus. Gappa calcule, vérifie et prouve formellement des bornes d'erreur pour des programmes numériques utilisant la virgule flottante. Après avoir écrit un code numérique, on le retranscrit dans Gappa et il calcule automatiquement une borne pour les erreurs d'arrondi. Cette borne est parfois assez large, mais on peut donner à Gappa des indications qui permettent de l'affiner. Gappa ne se contente pas de calculer une borne, il fournit aussi une preuve formelle dans le langage COQ [2] : cette preuve

1. distribué sous licences CeCILL et LGPL et disponible à l'adresse http://lipforge.ens-lyon.fr/www/gappa/

2. Ce logiciel permet d'exprimer et de vérifier des preuves formelles. Il est distribué sous licence LGPL et accessible à l'adresse http://coq.inria.fr/

formalise l'arithmétique flottante telle qu'elle est standardisée par la norme IEEE-754 et démontre que l'erreur est effectivement majorée par la borne donnée par Gappa.

La question de l'erreur d'approximation en revanche n'avait pas trouvé de solution totalement satisfaisante, à notre connaissance. En apparence, il s'agit d'un problème plus simple : alors que l'erreur due aux arrondis est de nature profondément discrète et présente un caractère chaotique en fonction du point d'évaluation $x$, l'erreur d'approximation est en général une fonction de $x$ très régulière (cf. figure A page 27) :

— pour l'erreur absolue, $\varepsilon(x) = |p(x) - f(x)|$ ;
— et pour l'erreur relative $\varepsilon(x) = |p(x)/f(x) - 1|$.

Si on omet la valeur absolue, il s'agit en général d'une fonction $\mathcal{C}^\infty$ et même analytique.

Calculer un majorant fin de l'erreur d'approximation sur l'intervalle, ce n'est rien d'autre que calculer un encadrement fin de la norme sup de la fonction $\varepsilon$ sur l'intervalle $[a, b]$ considéré : on cherche un intervalle $\boldsymbol{r}$ assez fin tel que $\|p - f\|_\infty \in \boldsymbol{r}$ ou $\|p/f - 1\|_\infty \in \boldsymbol{r}$, suivant qu'on s'intéresse à l'erreur absolue ou l'erreur relative. De façon équivalente, il suffit de calculer un encadrement fin du maximum et du minimum global de la fonction d'erreur $\varepsilon$. Dans toute la suite, nous nous limiterons donc à la recherche d'un encadrement fin du maximum d'une fonction de la forme $\varepsilon = p - f$ ou $\varepsilon = p/f - 1$ sur $[a, b]$ (en pratique, la recherche du minimum se fait simultanément). En outre, nous supposerons que $\varepsilon$ est une fonction suffisamment régulière : au moins de classe $\mathcal{C}^2$ sur $[a, b]$, mais de préférence $\mathcal{C}^\infty$. Cette hypothèse est toujours vérifiée pour les fonctions des libms.

Le problème de maximisation globale d'une fonction régulière univariée a déjà été étudié en détail mais, comme nous allons le voir, les instances qui nous intéressent ont des spécificités qui font échouer les méthodes traditionnelles.

## 3.2 Algorithmes classiques

### 3.2.1 Algorithmes purement numériques

La recherche des extrema locaux et globaux de fonctions représente, en soi, un domaine de recherche à part entière. La plupart des travaux se concentre sur le problème de fonctions à plusieurs variables [Kea96, HW04, RR88]. Dans le cas de fonctions à une seule variable, des algorithmes numériques efficaces existent : dichotomie, algorithme de Newton ou de la sécante pour trouver les zéros de la dérivée, ainsi que des algorithmes n'utilisant pas la dérivée [Bre02].

Dans Sollya, la commande `dirtyinfnorm` calcule la norme sup d'une fonction en utilisant un tel algorithme numérique certifié. Concrètement, pour calculer la norme d'une fonction $\varepsilon$ sur l'intervalle $[a, b]$, `dirtyinfnorm` prend $N$ points équirépartis dans l'intervalle

$$a = x_1 < x_2 < \cdots < x_{N-1} < x_N = b.$$

La valeur de $N$ est fixée par la variable globale `points`. La fonction $\varepsilon$ et sa dérivée $\varepsilon'$ sont évaluées en chacun des points $x_i$. Lorsque la dérivée change de signe entre deux $x_i$ successifs, l'algorithme de Newton est utilisé pour calculer une valeur $z$ telle que $\varepsilon'(z) \simeq 0$. Finalement, `dirtyinfnorm` renvoie le maximum des valeurs absolues des $\varepsilon(x_i)$ et des $\varepsilon(z)$.

> **Exemple 3.1.** *Prenons un exemple qui nous suivra tout au long de ce chapitre. On a approché la fonction* $f = \exp$ *sur* $[a, b] = [-1/4, 1/4]$ *en erreur absolue par un polynôme* $p$ *(ses coefficients sont en* `double`*; il a été obtenu en utilisant* `fpminimax`*). On peut estimer numériquement la norme sup de l'erreur avec la commande* `dirtyinfnorm`

```
> f=exp(x);
> a=-1/4; b=1/4;
> p = 562949953419691/2^49 + 4503599627451287/2^52 * x + 4503599640812691/2^53 *
 x^2 + 3002399712342729/2^54 * x^3 + 6004782173189321/2^57 * x^4 + 48038752142
11745/2^59 * x^5 + 8024226169454 71/2^59 * x^6 + 7307438221723147/2^65 * x^7;
> dirtyinfnorm(p-f,[a;b]);
3.790459993016317583936894120002714884474922313896e-12
```

De tels algorithmes ne peuvent convenir pour le problème qui nous intéresse. En effet, ils ne fournissent qu'une valeur approchée du maximum. Cette valeur approchée est obtenue en trouvant une valeur $z$ très proche d'un point où $\varepsilon$ atteint son maximum global. Ainsi, $\varepsilon(z)$ constitue une bonne approximation du maximum. Par construction, il s'agit en fait d'un minorant du maximum.

Cela étant, nous pouvons obtenir un encadrement fin du maximum en modifiant très légèrement ces méthodes. La commande `dirtyinfnorm`, par exemple, calcule finalement une liste $(z_1, \ldots, z_K)$ correspondant à des valeurs approchées de très bonne qualité de tous les zéros de la dérivée de $\varepsilon$. À partir de chaque point $z_i$, on peut construire un petit intervalle $\boldsymbol{z_i}$ de telle sorte qu'on soit sûr que le zéro de $\varepsilon'$ correspondant soit effectivement dedans.

**Exemple 3.2.** *Reprenons notre exemple. Dans Sollya, on peut calculer une liste des zéros approchés d'une fonction en utilisant la commande* `dirtyfindzeros`*. Cette commande fonctionne sur le même principe que* `dirtyinfnorm` *(découpage en sous-intervalles et utilisation de l'algorithme de Newton sur les intervalles où la fonction change de signe).*

```
> L=dirtyfindzeros(diff(p-f),[a;b]);
> L;
[|-0.231820309018269537323323397537789391629464767597925, -0.1796986308554787502
49578923880652153541867798442903, -0.100706619393275524950872904280396603331923 4
53857183, -5.966630638701319246606899026270325817133159140031 7e-3, 9.05197984640
162791089756476746056691703451724254 56e-2, 0.173729748084053447059233034422935 89
6907282213361598, 0.230071603383168325971876395988441115589773957487346|]
```

*Considérons le premier zéro : normalement, Sollya essaie de faire en sorte que ce soit une bonne valeur approchée à la précision courante (*`prec` $= 165$ *par défaut). Donc en arrondissant cette valeur vers le bas et vers le haut sur une précision de* $155$ *bits, nous devrions être à peu près sûrs d'obtenir des valeurs strictement inférieure et strictement supérieure au zéro exact.*

```
> z = L[0];
> zlower = round(z, 155, RD);
> zupper = round(z, 155, RU);
```

*Pour le vérifier, il suffit simplement d'évaluer la dérivée en utilisant l'arithmétique d'intervalle en* `zlower` *et en* `zupper`*.*

```
> midpointmode=on;
Midpoint mode has been activated.
> evaluate(diff(p-f), [zlower]);
0.92076474295644339578230837743062550862910151954 7~2/3~e-55
> evaluate(diff(p-f), [zupper]);
-0.61157237244036348668266045135939093668180713466 37~7/4~e-56
```

*L'arithmétique d'intervalle nous permet d'avoir une valeur certifiée. En l'occurrence, nous en déduisons de façon certaine que le signe de $(p-f)'$ a changé entre* `zlower` *et* `zupper` *: par continuité de $(p-f)'$, on en déduit que le zéro de $(p-f)'$ est bien compris dans l'intervalle* [`zlower`, `zupper`]*. En évaluant $p-f$ sur l'intervalle* [`zlower`, `zupper`]*, on obtient un bon encadrement de l'extremum local correspondant :*

```
> evaluate(p-f, [zlower, zupper]);
Midpoint mode has been activated.
0.23410576803476360322428092483867992l1~3/5~e-11
```

On peut ainsi obtenir très facilement un encadrement $\boldsymbol{z_i}$ de chaque zéro de $\varepsilon'$. On peut alors évaluer $\varepsilon$ sur chacun de ces intervalles : cela donne une liste d'intervalles $\boldsymbol{y_i}$. Si $\varepsilon$ est dérivable sur $[a, b]$, le maximum global est atteint en $a$, en $b$ ou bien en un point où la dérivée s'annule. Notons $\boldsymbol{y_0}$ un intervalle englobant $\varepsilon(a)$ et $\boldsymbol{y_{K+1}}$ un intervalle englobant $\varepsilon(b)$. Le maximum global est donc compris dans l'un des $\boldsymbol{y_i}$.

**Exemple 3.3.** *Nous pouvons conclure notre exemple (les opérateurs* `.:` *et* `:.` *servent à ajouter un élément respectivement en tête et en queue de liste). Cette fois, nous obtenons bien un encadrement du maximum global de $\varepsilon$ sur $[a, b]$ (pour un encadrement de la norme sup, il aurait suffi de calculer le minimum global en même temps).*

```
> L = a .: L :. b;
Midpoint mode has been activated.
> Lyi = [||];
> for z in L do {
    zlower = round(z, 155, RD);
    zupper = round(z, 155, RU);
    yi = evaluate(p-f, [zlower, zupper]);
    Lyi = Lyi :. yi;
  };
>
> maxlower = 0;
> maxupper = 0;
> for yi in Lyi do {
    if ( inf(yi) > maxlower ) then maxlower = inf(yi);
    if ( sup(yi) > maxupper ) then maxupper = sup(yi);
  };
>
> [maxlower; maxupper];
0.37153774362304950604936487265073463l9~5/7~e-11
```

Mais cet encadrement est-il vraiment certifié ? Tout repose sur la recherche des zéros par `dirtyfindzeros` : cet algorithme donne de très bons résultats en pratique, mais il ne garantit absolument pas que tous les zéros de $\varepsilon'$ soient bien dans la liste. Rien ne dit qu'on n'a pas oublié un extremum au passage, qui se révélerait être le vrai maximum global. Même si cet encadrement a de très bonnes chances d'être correct, nous ne pouvons le considérer comme un résultat certifié.

**Remarque 3.4.** *La borne* `maxupper` *n'est donc pas un majorant certifié du maximum de* $\varepsilon$*. Cependant, la borne* `maxlower` *constitue effectivement un minorant du maximum : en effet, nous sommes sûrs, par construction que* [`maxlower`, `maxupper`] *encadre un extremum local de* $\varepsilon$*.*
*La vraie difficulté consiste donc à trouver un* ***majorant certifié*** *du maximum de* $\varepsilon$*.*

### 3.2.2 Utilisation de l'arithmétique d'intervalle

Il faut donc se détourner des algorithmes purement numériques et chercher des méthodes permettant de garantir la fiabilité du calcul. Dans toute la suite, nous supposerons que la fonction $\varepsilon$ dont on souhaite connaître le maximum est donnée sous la forme d'une expression faisant intervenir des variables, des constantes et des fonctions de base (addition, multiplication, exponentielle, fonctions trigonométriques, etc.) C'est exactement la situation dans laquelle on se trouve lorsqu'on utilise Sollya par exemple.

Une première possibilité consiste à utiliser directement l'arithmétique d'intervalle : en évaluant $\varepsilon$ par arithmétique d'intervalle sur $[a, b]$, on obtient un intervalle $J$ qui, par définition, vérifie $\varepsilon([a, b]) \subseteq J$. La borne supérieure de $J$ donne bien un majorant du maximum.

**Exemple 3.5.** *Si on applique cette stratégie sur notre exemple, on obtient*

$$J = [-0,5366377335,\ 0,5052246337]\,.$$

*Évidemment, on a bien un majorant, mais on est très loin de la valeur approximative* $3,7\,e{-}12$ *qu'on désirerait obtenir. Ce résultat est extrêmement surestimé du fait du phénomène de décorrélation qui se produit lorsqu'on utilise l'arithmétique d'intervalle sur des expressions composées et des intervalles assez larges (nous avons décrit ce phénomène en page 71).*

En général, lorsque la taille de l'intervalle $[a, b]$ diminue, les effets de décorrélation diminuent. On peut donc adopter une approche dichotomique : prendre $c$ dans l'intervalle $[a, b]$ (par exemple le milieu, mais pas forcément) et évaluer $\varepsilon$ par arithmétique d'intervalle sur $[a, c]$ et sur $[c, b]$ séparément. On obtient ainsi deux intervalles $J_1$ et $J_2$ tels que, par construction,

$$\varphi([a, b]) \subseteq J_1 \cup J_2.$$

**Exemple 3.6.** *Si on applique cette stratégie à notre exemple (avec* $c = 0$*), on obtient :*

$$J_1 = [-0,2526123169,\ 0,2526123169]$$
$$J_2 = [-0,2840254167,\ 0,2840254167]\,.$$

*Cela nous donne le majorant* $0,2840254167$ *qui est effectivement à peu près deux fois meilleur. On est cependant encore très loin d'avoir une borne fine.*

Si la fonction $\varepsilon$ est de classe $\mathcal{C}^1$ sur $[a, b]$ et si on pense que $\varepsilon$ est monotone sur un sous-intervalle $I$, on peut essayer d'utiliser cette information. Puisque la fonction $\varepsilon$ est donnée par une expression, on peut calculer formellement une expression de sa dérivée $\varepsilon'$ et évaluer $\varepsilon'$ par arithmétique d'intervalle sur $I$. Soit $J'$ l'intervalle obtenu. Si 0 n'appartient pas à $J'$, et puisque

$\varepsilon'([a,\, b]) \subseteq J'$, il est certain que $\varepsilon'$ ne s'annule pas sur $I$ et donc que $\varepsilon$ est monotone sur $[a,\, b]$. Alors, $\varepsilon([a,\, b]) = [\varepsilon(a),\, \varepsilon(b)]$ ou le contraire, suivant que $\varepsilon$ est croissante ou décroissante.

On peut évaluer $\varepsilon$ sur l'intervalle-point $[a,\, a]$ et sur l'intervalle-point $[b,\, b]$. Dans ce cas, on peut obtenir des encadrement arbitrairement précis de $\varepsilon(a)$ et $\varepsilon(b)$ en augmentant la précision des calculs (voir la discussion à ce sujet en page 71). On obtient donc un intervalle $\boldsymbol{y_a}$ tel que $\varepsilon(a) \in \boldsymbol{y_a}$ (et de même $\boldsymbol{y_b}$). Notons $\underline{\boldsymbol{y_a}}$ la borne inférieure de $\boldsymbol{y_a}$, et $\overline{\boldsymbol{y_b}}$ la borne supérieure de $\boldsymbol{y_b}$. On a alors

$$\varepsilon([a,\, b]) \subseteq \left[\underline{\boldsymbol{y_a}},\, \overline{\boldsymbol{y_b}}\right].$$

L'avantage, dans ce cas, est que l'intervalle englobant finalement obtenu peut être rendu arbitrairement fin en augmentant simplement la précision dans le calcul de $\boldsymbol{y_a}$ et $\boldsymbol{y_b}$.

Pour que cette méthode fonctionne, il faut que 0 n'appartienne pas à $J'$. Si l'évaluation de $\varepsilon'$ par intervalle est trop grossière, $J$ risque de contenir 0 alors même que $\varepsilon'$ ne s'annule pas sur l'intervalle $I$.

**Exemple 3.7.** *Sur l'intervalle* $[0,04,\, 0,05]$*, la fonction* $\varepsilon$ *de notre exemple semble croissante (voir figure 3.1). Pourtant, l'évaluation par intervalle de* $\varepsilon'$ *sur cet intervalle donne*

$$\varepsilon'([0,04,\, 0,05]) \subseteq J = [-1,046032209\, e{-}2,\, 1,046032229\, e{-}2]\,.$$

*Le phénomène de décorrélation est présent aussi lors de l'évaluation de* $\varepsilon'$ *et* $J$ *contient donc* 0 *alors que* $\varepsilon'$ *ne s'annule pas dans l'intervalle.*

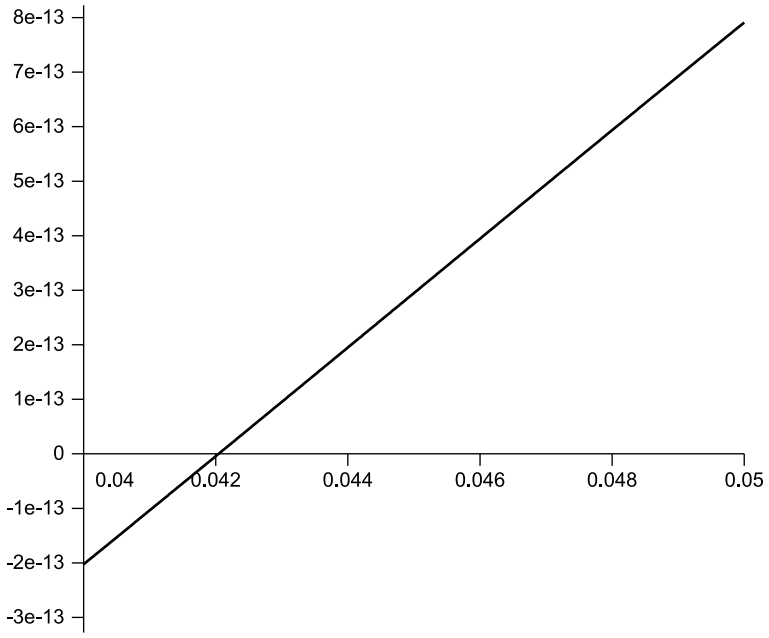



**Figure 3.1 :** Graphe de l'erreur $\varepsilon = p - f$ (données de l'exemple 3.1) sur l'intervalle $[0,04,\, 0,05]$. La fonction est manifestement croissante.

Ce principe d'évaluation de la dérivée pour établir la monotonie de $\varepsilon$ peut évidemment être poussé aux dérivées d'ordre supérieur si $\varepsilon$ est suffisamment régulière. L'expression de $\varepsilon'$ est généralement plus compliquée que l'expression de $\varepsilon$ (qu'on songe à la formule de dérivation d'un produit $(uv)' = u'v + v'u$ par exemple), ce qui rend la méthode peu intéressante aux ordres supérieurs. Mais dans certains cas (si $\varepsilon$ est l'addition d'un polynôme et d'une fonction simple comme

exp ou sin par exemple) les expressions des dérivées successives restent simples et peuvent être exploitées jusqu'à un ordre élevé.

Il est également possible d'exploiter $\varepsilon'$ pour améliorer l'évaluation de $\varepsilon$ sur l'intervalle $[a, b]$ en utilisant le théorème des accroissements finis :

$$\forall x_0 \in [a, b]\,, \forall x \in [a, b]\,, \exists\, \xi \in [a, b]\,, \quad \text{tel que} \quad \varepsilon(x) = \varepsilon(x_0) + (x - x_0)\varepsilon'(\xi).$$

Par conséquent l'évaluation par l'arithmétique d'intervalle de la formule

$$\varepsilon([x_0, x_0]) + [a - x_0, b - x_0]\, \varepsilon'([a, b])$$

donne un intervalle $J$ tel que $\varepsilon([a, b]) \subseteq J$.

**Exemple 3.8.** *Si l'on applique cette formule dans notre exemple, sur l'intervalle* $[a, b]$ *avec* $x_0 = 0$*, on obtient*

$$J = [-0,1341594332,\, 0,1341594332]\,.$$

*C'est bien mieux que ce qu'on obtenait par évaluation directe. Cependant, là encore, on est loin d'obtenir un encadrement fin.*

Cette méthode peut être étendue aux ordres supérieurs si la fonction $\varepsilon$ est suffisamment régulière puisque le théorème des accroissements finis se généralise avec la formule de Taylor :

$$\forall x_0, x \in [a, b]\,, \exists\, \xi \in [a, b]\,, \quad \text{tel que} \quad \varepsilon(x) = \left(\sum_{i=0}^{n-1} \frac{\varepsilon^{(i)}(x_0)}{i!}(x - x_0)^i\right) + \frac{\varepsilon^{(n)}(\xi)}{n!}(x - x_0)^n.$$

Évidemment, la remarque précédente vaut toujours : à moins que $\varepsilon$ soit une expression très simple, les dérivées successives sont des expressions de plus en plus compliquées et le procédé cesse d'être intéressant au-delà d'un certain ordre.

Toutes ces techniques peuvent évidemment être combinées les unes avec les autres pour fournir un algorithme d'évaluation par arithmétique d'intervalle plus performant que la méthode directe par composition des opérations de base. Lorsqu'on cherche à évaluer une expression composée $g \circ f$, il est également possible d'utiliser ces stratégies sur les sous-expressions plutôt que sur l'expression elle-même. La commande `evaluate` de Sollya utilise ces différentes astuces de façon heuristique ; son fonctionnement est décrit dans ses grandes lignes dans [CL07]. Elle dépend d'un certain nombre de variables globales qui permettent par exemple de fixer l'ordre utilisé dans la formule des accroissements finis.

**Exemple 3.9.** *Nous pouvons utiliser la commande* `evaluate` *de Sollya pour avoir un meilleur encadrement de l'image de* $\varepsilon$ *sur* $[a, b]$*. En jouant sur la variable* `taylorrecursions`*, on peut augmenter l'ordre dans la formule des accroissements finis.*

```
> evaluate(p-f, [a,b]);
[-3.55031720659761511760499084387177602778381535 9746e-2;3.55031720602172023027142964322322980891174504 7246e-2]
> taylorrecursions=5;
The number of recursions for Taylor evaluation has been set to 5.
> evaluate(p-f, [a,b]);
[-1.80039219485837050661926591642838018484325301451 25e-5;1.80039161896348317305806526788216131277294051 45125e-5]
```

*C'est beaucoup mieux, mais ce n'est pas encore suffisant pour calculer un majorant fin de* $\|\varepsilon\|_\infty$.

La commande `evaluate` est capable d'évaluer des fonctions comme $\sin(x)/\log(1+x)$ qui présente une fausse singularité en un point : pour cela, la fausse singularité est d'abord détectée de manière heuristique, puis une variante de la règle de l'Hôpital valable en arithmétique d'intervalle est utilisée. La description de cette méthode figure dans la version longue de [CL07].

### 3.2.3 Algorithme de Hansen

Puisque l'arithmétique d'intervalle seule ne permet pas, en général, de déterminer précisément le maximum d'une fonction sur un intervalle, des algorithmes spécifiques ont été conçus dans ce but. L'article fondateur sur ce sujet semble être celui de Hansen [Han79]. La plupart des algorithmes qui ont suivi sont des variantes de l'algorithme original de Hansen [Mes97].

Le principe de l'algorithme de Hansen est le suivant. On suppose que la fonction $\varepsilon$ est au moins de classe $\mathcal{C}^2$. Pour localiser le maximum global de $\varepsilon$ sur $[a, b]$, on applique une stratégie de type *branch-and-bound*. Une liste $\mathcal{L}$ est maintenue à jour ; cette liste contient des intervalles $I$ où il est possible que $\varepsilon$ atteigne son maximum. Au fur et à mesure, on diminue la taille ou le nombre des intervalles contenus dans la liste. On maintient aussi un minorant $M$ du maximum.

**Début de l'algorithme :** initialement, $\mathcal{L}$ contient seulement l'intervalle $[a, b]$ et

$$M = \max\{\varepsilon(a), \varepsilon(b)\}.$$

Plus exactement, on évalue $\varepsilon$ sur les intervalles-point $[a, a]$ et $[b, b]$ : on peut obtenir ainsi une minoration aussi précise qu'on le souhaite des valeurs $\varepsilon(a)$ et $\varepsilon(b)$.

**Suppression d'intervalles :**

1. Au cours de l'algorithme, si $I$ est un intervalle de $\mathcal{L}$, notons $J$ le résultat d'une évaluation par arithmétique d'intervalle de $\varepsilon$ sur $I$. Si la borne supérieure $\overline{J}$ de $J$ est inférieure à $M$, on peut retirer $I$ de la liste. À l'inverse, si la borne inférieure $\underline{J}$ de $J$ est supérieure à $M$, on peut poser $M = \underline{J}$.
2. Si l'évaluation de $\varepsilon'$ sur un intervalle $I$ de $\mathcal{L}$ donne un intervalle $J'$ et que $0 \notin J'$, on peut retirer l'intervalle $I$ de la liste.
3. Si l'évaluation de $\varepsilon''$ sur un intervalle $I$ de $\mathcal{L}$ donne un intervalle $J''$ ne contenant que des nombres strictement positifs, la fonction $\varepsilon$ est convexe sur $I$. On peut alors retirer $I$ de la liste.

**Diminution de la taille des intervalles :**

1. Notons $I$ un intervalle de la liste. Si $\varepsilon'$ s'annule en un point $x$ de $I$, le théorème des accroissements finis donne

$$\forall x_0 \in I,\ \exists \xi \in I,\ \varepsilon'(x) = 0 = \varepsilon'(x_0) + (x - x_0)\varepsilon''(\xi).$$

Choisissons $x_0$ dans $I$ (par exemple le milieu) et notons $N(I)$ une évaluation par arithmétique d'intervalle de l'expression

$$[x_0, x_0] - \frac{\varepsilon'([x_0, x_0])}{\varepsilon''(I)}.$$

Alors, l'ensemble $S$ des points de $I$ où $\varepsilon'$ s'annule vérifie $S \subseteq N(I) \cap I$. Il s'agit de la *méthode de Newton par intervalle*. Dans le cas où $\varepsilon''$ s'annule sur $I$, la méthode fonctionne aussi, en utilisant

$$\varepsilon''(I) \subseteq [u, v] \quad \Longrightarrow \quad \frac{1}{\varepsilon''(I)} \subseteq \left]-\infty, \frac{1}{v}\right] \cup \left[\frac{1}{u}, +\infty\right[.$$

En appliquant la méthode de Newton à l'intervalle $I$, on peut donc diminuer la taille de $I$ (voire le supprimer complètement) ou bien le couper en deux intervalles $I_1$ et $I_2$ tels que $I_1 \cup I_2 \subseteq I$.

2. Puisque $M$ est un minorant du maximum global, on peut éliminer tous les points de $I$ pour lesquels $\varepsilon(x) \leq M$. Choisissons $x_0 \in I$ (le milieu par exemple). La formule de Taylor à l'ordre 2 indique

$$\forall x \in I,\ \exists \xi \in I,\ \varepsilon(x) = \varepsilon(x_0) + (x - x_0)\varepsilon'(x_0) + \frac{(x - x_0)^2}{2}\,\varepsilon''(\xi).$$

Par conséquent, si pour tout $z \in \varepsilon''(I)$ on a

$$\varepsilon(x_0) + (x - x_0)\varepsilon'(x_0) + \frac{(x - x_0)^2}{2}\,z \ \leq\ M,$$

on est sûr que $\varepsilon(x) \leq M$. Notons $[u, v]$ le résultat d'une évaluation par arithmétique d'intervalle de $\varepsilon''(I)$. Pour que $\varepsilon(x) \leq M$, il est donc suffisant que

$$(\varepsilon(x_0) - M) + (x - x_0)\varepsilon'(x_0) + \frac{(x - x_0)^2}{2}\,v \ \leq\ 0. \tag{3.1}$$

Cette inéquation du second degré peut être résolue exactement. De la même façon qu'avec la méthode de Newton, on pourra, suivant les cas, diminuer la taille de $I$ (voire le supprimer) ou le scinder en deux sous-intervalles $I_1$ et $I_2$ tels que $I_1 \cup I_2 \subseteq I$.

L'algorithme s'arrête lorsque les intervalles de la liste $\mathcal{L}$ sont jugés suffisamment fins. On peut alors évaluer $\varepsilon$ sur chacun des intervalles $I$ de $\mathcal{L}$ : on obtient ainsi un minorant `maxlower` et un majorant `maxupper` du maximum, de la même façon que dans l'exemple 3.3.

Une autre possibilité consiste à choisir un paramètre $\delta > 0$ et à remplacer $M$ par $M + \delta$ dans l'inéquation (3.1). On élimine ainsi, au cours de l'algorithme, tous les points $x$ de $[a, b]$ pour lesquels $\varepsilon(x) \geq M + \delta$. L'algorithme s'arrête lorsque la liste $\mathcal{L}$ ne contient plus aucun intervalle : on sait alors que le maximum de $\varepsilon$ est compris entre $M$ et $M + \delta$.

Nous avons essayé d'utiliser un algorithme de ce type pour calculer un encadrement certifié de la norme sup de fonctions d'erreur. Dans l'algorithme, nous n'évaluons pas les expressions en composant simplement les fonctions par intervalle : nous utilisons un algorithme d'évaluation perfectionné (l'équivalent de la commande `evaluate` de Sollya) en nous appuyant sur les remarques de la section 3.2.2. La commande `infnorm` de Sollya est une implantation de cet algorithme (l'algorithme lui-même est décrit dans [CL07]).

**Exemple 3.10.** *Nous terminons notre exemple fil rouge en utilisant la commande* `infnorm` *de Sollya qui renvoie un intervalle certifié encadrant la norme sup d'une fonction (à ne pas confondre avec* `dirtyinfnorm` *qui se contente d'estimer numériquement cette norme).*

```
> dirtyinfnorm(p-f, [a,b]);
3.790459993016317583936894120002714884474922313896e-12
> infnorm(p-f, [a,b]);
0.37904599930163175839368941200027148844~4/9~e-11
```

Sur cet exemple, la commande `infnorm` fonctionne très bien : l'encadrement calculé est très fin et il est obtenu dans un temps raisonnable (environ une seconde sur un processeur cadencé

à 3 GHz). Mais il ne s'agit que d'un exemple assez simple : un polynôme de degré 7 approchant une fonction $f$ en erreur absolue. Dans le cadre du développement de `CRlibm`, il est courant de manipuler des polynômes de degré 20 et on considère généralement l'erreur relative $\varepsilon = p/f - 1$. Le cas de l'erreur relative est plus difficile en général car les expressions des dérivées première et seconde de $\varepsilon$ sont bien plus compliquées pour l'erreur relative que pour l'erreur absolue, ce qui augmente l'importance du phénomène de décorrélation.

Comme nous avons pu le voir en section 3.2.2, l'évaluation $\varepsilon$ sur un intervalle $I$ donne des résultats très grossiers, à moins que $I$ soit minuscule. En employant des algorithmes sophistiqués comme la commande `evaluate` de Sollya, on obtient un intervalle de meilleure qualité, mais la surestimation par rapport à l'intervalle image exact demeure très importante.

Les algorithmes du type *algorithme de Hansen* reposent sur le fait que, lorsque $I$ n'est pas trop grand, l'évaluation de $\varepsilon$, $\varepsilon'$ et $\varepsilon''$ sur $I$ fournit des intervalles $J$, $J'$ et $J''$ pertinents. Ils n'ont pas besoin d'être très fins, mais s'ils surestiment trop grossièrement les images exactes, ils ne permettent ni de supprimer, ni de réduire la taille des intervalles de la liste $\mathcal{L}$.

Ces algorithmes se comportent extrêmement mal dans le cas qui nous intéresse, du fait de la forme très particulière de $\varepsilon$. Pour obtenir un majorant de la norme sup avec seulement un ou deux chiffres corrects, le temps de calcul nécessaire croît extrêmement vite lorsque le degré du polynôme d'approximation augmente. De même, la quantité de mémoire nécessaire (due au nombre d'intervalles contenus dans la liste $\mathcal{L}$) augmente très rapidement, de sorte qu'il n'est généralement pas possible, en pratique, de traiter certaines des erreurs qu'on rencontre dans le développement de `CRlibm`.

### 3.2.4 Particularité de notre problème

Pour comprendre ce problème, intéressons-nous à la taille que doivent avoir les intervalles $I$ pour que l'arithmétique d'intervalle donne des résultats pertinents. Considérons par exemple le cas de l'erreur absolue.

Pour éliminer $I$ ou en réduire la taille durant l'algorithme, on a recours à une évaluation par arithmétique d'intervalle de $\varepsilon$, $\varepsilon'$, $\varepsilon''$, etc. Or, du point de vue de l'arithmétique d'intervalle, $\varepsilon(I) = (p-f)(I)$ est interprété comme $p(I)-f(I)$ : c'est le phénomène de décorrélation que nous avons déjà analysé précédemment. L'arithmétique d'intervalle ignore le fait que la valeur de $p$ est fortement corrélée à celle de $f$ et elle les traite comme deux données indépendantes.

Ce phénomène se retrouve également aux ordres supérieurs (en s'atténuant peu à peu) : en effet, lorsque $p$ est proche d'être un polynôme de meilleure approximation de $f$, l'erreur $\varepsilon$ oscille au moins $n+2$ fois (pas tout à fait parfaitement) entre ses valeurs extrêmes. Dans ce cas $\varepsilon' = p' - f'$ oscille au moins $n+1$ fois (mais encore moins parfaitement) entre ses valeurs extrêmes. Donc, d'après le théorème de La Vallée Poussin (théorème 2.7 page 83), $p'$ n'est pas très loin d'être le polynôme de meilleure approximation de $f'$. Et ainsi de suite aux ordres supérieurs. Par exemple, pour notre exemple fil rouge, on a

$$\begin{aligned} \|\varepsilon\|_\infty &\simeq 3,8\,\mathrm{e}{-12}, \\ \|\varepsilon'\|_\infty &\simeq 9,2\,\mathrm{e}{-10}, \\ \|\varepsilon''\|_\infty &\simeq 7,4\,\mathrm{e}{-8}. \end{aligned}$$

Considérons deux intervalles $[u,\, u+\delta]$ et $[v,\, v+\delta']$ (avec $\delta \simeq \delta'$). Lorsqu'on soustrait le second du premier on obtient

$$[u-v-\delta',\, u-v+\delta] :$$

c'est un intervalle centré approximativement en $(u-v)$ et de taille $\delta+\delta' \simeq 2\delta$.

Deux cas distincts se présentent :

— si $(u-v) \ll \delta$, on a quasiment l'intervalle $[-\delta',\, \delta]$ ;

— si $\delta \ll (u - v)$, on a quasiment l'intervalle-point $[u - v,\, u - v]$.

Appliquons ce constat à la situation qui nous intéresse : on se donne un intervalle $I$ et on voudrait quantifier l'effet de décorrélation qu'il y a entre l'image exacte $(p - f)(I)$ et la différence calculée $p(I) - f(I)$. Nous supposerons pour simplifier que les intervalles $p(I)$ et $f(I)$ ont été calculés exactement.

Par définition, l'ordre de grandeur des nombres compris dans $(p-f)(I)$ est inférieur à $\|p - f\|_\infty = \|\varepsilon\|_\infty$. Dans le cadre de l'implémentation de fonctions avec arrondi correct, il est courant d'avoir

$$\|\varepsilon\|_\infty \simeq 2^{-120}.$$

Si $I$ est déjà assez petit, on peut supposer que $p$ et $f$ ont un comportement quasiment linéaire sur l'intervalle $I$ : dans ce cas, on a approximativement $p(I) \simeq [u,\, u + \delta]$, avec $u = p(\underline{I})$ et

$$\delta \simeq p'(\underline{I})\ \mathrm{diam}(I),$$

où $\underline{I}$ désigne la borne inférieure de $I$ et $\mathrm{diam}(I)$ son diamètre. En supposant que $p'$ est de l'ordre de l'unité sur $I$, l'ordre de grandeur de $\delta$ est donc $\mathrm{diam}(I)$. De même, $f(I) \simeq [v,\, v + \delta']$, où $v = f(\underline{I})$ et $\delta'$ est aussi de l'ordre de grandeur de $\mathrm{diam}(I)$. On peut estimer l'ordre de grandeur de $p(\underline{I}) - f(\underline{I})$ à $\|p - f\|_\infty$.

Nous avons donc deux cas possibles :

— si $\|\varepsilon\|_\infty \ll \mathrm{diam}(I)$, la soustraction des intervalles donne quasiment $[-\mathrm{diam}(I),\, \mathrm{diam}(I)]$ ;

— si $\mathrm{diam}(I) \ll \|\varepsilon\|_\infty$, la soustraction donne un intervalle très fin autour de $p(\underline{I}) - f(\underline{I})$, autrement dit, quelque chose de représentatif du véritable intervalle image $(p - f)(I)$.

Autrement dit, on ne peut rien tirer d'intéressant de l'arithmétique d'intervalle **tant qu'on essaie de l'utiliser avec des intervalles dont le diamètre est supérieur à** $\|\varepsilon\|_\infty$. Dans l'algorithme de Hansen, les intervalles sont donc subdivisés en intervalles plus petits jusqu'à obtenir des intervalles de taille environ $\|\varepsilon\|_\infty$ : ce n'est qu'à partir de ce moment que l'algorithme parvient à supprimer des intervalles de la liste. Pour les fonctions d'erreur $\varepsilon$ rencontrées dans `CRlibm`, il faut donc s'attendre à environ $2^{120}$ intervalles, ce qui explique l'échec complet de ce genre d'algorithmes. L'utilisation de la commande `evaluate` de Sollya améliore un peu les choses, mais ne peut pas faire de miracle.

Pour contourner ce problème, il faut donc impérativement supprimer *a priori* la corrélation qu'il y a entre $p$ et $f$ dans l'expression de $\varepsilon$. Cette « recorrélation » consiste à récrire $p - f$ sous une forme qui ne fasse plus apparaître de différence entre deux termes ayant le même ordre de grandeur.

## 3.3 Une nouvelle approche

Imaginons un instant que $f$ est donnée par une série entière. Notons $f = \sum_{i=0}^{+\infty} f_i x^i$ et $p = \sum_{i=0}^{n} p_i x^i$. Alors, $f - p$ est égal à la série

$$\left(\sum_{i=0}^{n} (f_i - p_i) x^i\right) + \left(\sum_{i=n+1}^{+\infty} f_i x^i\right).$$

Comme $p$ approche $f$, $p'$ approche $f'$, etc., les coefficients $p_i$ et $f_i$ $(i \leq n)$ sont certainement assez proches et les soustractions $f_i - p_i$ cancellent. La somme

$$\sum_{i=0}^{n} (f_i - p_i) x^i$$

capture donc le phénomène de corrélation entre $p$ et $f$. En d'autre termes, nous avons remplacé le phénomène de corrélation entre $p$ et $f$ sur l'intervalle $[a, b]$ par un phénomène de cancellation entre les coefficients $p_i$ et les coefficients $f_i$. Ce phénomène de cancellation est bénin puisqu'il suffit d'effectuer la soustraction avec suffisamment de précision pour le rendre inoffensif.

Il n'est pas réellement nécessaire que $f$ soit donnée par une série : ce dont on a besoin, c'est de pouvoir écrire $f$ sous la forme $T + R$, où $T$ est un polynôme tel que le reste $R$ soit négligeable devant $\|\varepsilon\|_\infty$ (le degré de $T$ doit donc être assez nettement supérieur au degré de $p$). Au premier abord, il semble qu'on n'ait rien fait d'autre que déplacer le problème : pour borner de façon certifiée l'erreur entre une approximation $p$ (le plus souvent un polynôme) et la fonction $f$, on a besoin de trouver un polynôme $T$ qui approche $f$ correctement, et de borner de façon certifiée l'erreur $R = f - T$. En réalité, le problème, s'il n'a pas changé dans sa nature, a changé dans sa forme :

— d'une part, nous avons le choix du polynôme $T$, ce qui nous autorise donc à choisir des polynômes particuliers pour lesquels une majoration analytique du reste est connue ;

— d'autre part, nous n'avons pas besoin de borner finement l'erreur : nous pouvons nous permettre de prendre un polynôme $T$ de degré suffisamment élevé pour que la borne d'erreur, même si elle est surestimée d'un gros facteur, reste inférieure à $\|\varepsilon\|_\infty$. Par contre, l'erreur entre $p$ et $f$ doit être bornée très finement.

### 3.3.1 Cas de l'erreur absolue

Le cas où $\varepsilon$ est une erreur absolue apparaît donc assez simple : on commence par calculer un polynôme $T$ qui approche $f$. On calcule en même temps une borne $\theta$ telle que $\|T - f\|_\infty \leq \theta$. Bien sûr, il faut que $\theta \ll \|\varepsilon\|_\infty$.

On a alors, par inégalité triangulaire :

$$\|\varepsilon\|_\infty = \|p - f\|_\infty \leq \|p - T\|_\infty + \|T - f\|_\infty \leq \|p - T\|_\infty + \theta.$$

Il suffit maintenant de calculer un encadrement fin de $\|p - T\|_\infty$. Évidemment, on ne laisse pas $p-T$ écrit sous cette forme, mais on calcule explicitement les coefficients de $p-T$. Les cancellations ont lieu lors du calcul de ces coefficients et le polynôme ainsi obtenu ne fait plus apparaître de problème de décorrélation. On peut alors utiliser l'algorithme de Hansen (par exemple) pour calculer un encadrement certifié $\|p - T\|_\infty$. On peut aussi s'appuyer sur le fait que $p - T$ est un polynôme pour employer des algorithmes *ad hoc* plus efficaces (voir section 3.5.1).

### 3.3.2 Cas de l'erreur relative

Pour l'erreur relative, la situation est un peu plus délicate. Nous nous appuyons sur la remarque suivante : dans l'exemple 3.3, la méthode que nous avons utilisée donnait de très bons résultats pratiques, pour un surcoût modique par rapport à un algorithme non certifié. Nous recherchions les zéros de $\varepsilon'$ à l'aide d'un algorithme non certifié (mais de bonne qualité) ; ensuite, chaque zéro $z_i$ était encadré de façon rigoureuse dans un intervalle $\boldsymbol{z_i}$ (en utilisant l'arithmétique d'intervalle sur des intervalles-point pour vérifier que $\varepsilon'$ changeait bien de signe dans l'intervalle). Le diamètre de $\boldsymbol{z_i}$ peut être rendu arbitrairement petit en approchant numériquement $z_i$ avec plus de précision. Enfin, une simple évaluation de $\varepsilon$ sur chacun des $\boldsymbol{z_i}$ permettait de conclure. Cette évaluation de $\varepsilon$ sur les intervalles $\boldsymbol{z_i}$ ne posait pas de problème de décorrélation car $\operatorname{diam}(\boldsymbol{z_i})$ était beaucoup plus petit que $\|\varepsilon\|_\infty$. Il ne manquait qu'une seule chose pour que cet algorithme soit parfaitement rigoureux : certifier qu'aucun zéro de $\varepsilon'$ n'était oublié dans la liste des $z_i$.

Imaginons que $\varepsilon'$ soit un polynôme : dans ce cas, le nombre de zéros de $\varepsilon'$ dans l'intervalle $[a, b]$ peut être déterminé rigoureusement. Nous verrons en section 3.5.1 des techniques permettant de localiser de façon certifiée toutes les racines d'un polynôme dans un intervalle.

Évidemment $\varepsilon'$ n'est pas un polynôme, mais là encore, on peut l'approcher de très près par un polynôme $T$. Les zéros de $\varepsilon'$ ne peuvent être loin des zéros de $T$ : nous verrons comment formaliser ce raisonnement en section 3.5.2.

L'expression de la dérivée de l'erreur relative est

$$\varepsilon' = \frac{p'f - f'p}{f^2}.$$

Si $f$ a un zéro dans l'intervalle $[a, b]$, la fonction $(p'f - f'p)/f^2$ n'est définie que par prolongement continu en ce zéro. Ceci rend le calcul automatique du polynôme $T$ difficile. Cependant, ce qui nous intéresse ici, ce sont les zéros de $(p'f - f'p)/f^2$. Ce sont aussi des zéros de $p'f - f'p$. Nous choisissons donc plutôt un polynôme $T$ qui approche $p'f - f'p$. Ce faisant, on ajoute peut-être des zéros artificiels, mais qui n'auront aucune contribution lors de l'évaluation finale de la fonction d'erreur et qui sont donc sans importance.

À présent, il nous faut à présent voir comment faire pour calculer le polynôme $T$ en même temps que sa borne d'erreur $\theta$ de façon rigoureuse. Ce sera l'objet de la prochaine section. Nous reviendrons ensuite sur la localisation des zéros d'un polynôme (section 3.5.1) et sur la technique qui permet de passer des zéros d'un polynôme aux zéros d'une fonction quelconque (section 3.5.2).

## 3.4 Calcul du polynôme $T$

Nous nous posons le problème général suivant : soit $\tau$ une fonction suffisamment régulière et une borne $\theta > 0$, calculer un polynôme $T$ qui approche la fonction $\tau$ avec une erreur bornée par $\theta$. La méthode qui vient sans doute le plus naturellement à l'esprit (qu'on repense à la notion de développement en série qui nous a servi de cadre informel un peu plus haut) est l'utilisation d'un développement de Taylor tronqué.

Si $\tau$ est $n$ fois différentiable sur l'intervalle $[a, b]$ et si $x_0 \in [a, b]$, la formule de Taylor à l'ordre $n$ affirme que, pour tout $x$ dans $[a, b]$,

$$\tau(x) = \sum_{i=0}^{n-1} \frac{\tau^{(i)}(x_0)}{i!} (x - x_0)^i + \Delta_n(x, \xi), \quad \text{avec} \quad \Delta_n(x, \xi) = \frac{\tau^{(n)}(\xi)(x - x_0)^n}{n!}, \tag{3.2}$$

où $\tau^{(i)}$ désigne la dérivée $i$-ème de $\tau$ et $\xi$ est un réel compris strictement entre $x$ et $x_0$.

L'utilisation pratique de cette formule pose deux problèmes :

— il faut calculer les dérivées successives de $\tau$ au point $x_0$. Outre le calcul proprement dit, il faut remarquer que les valeurs $\tau^{(i)}(x_0)$ n'ont aucune raison d'être exactement représentables. Donc en pratique, il faudra les arrondir et ajouter cette erreur induite dans la borne d'erreur ;

— de plus, il faut calculer un majorant de $\Delta_n(x, \xi)$. Il n'est pas nécessaire que ce majorant soit très fin, mais il faut qu'il soit rigoureusement obtenu.

### 3.4.1 Techniques de différentiation automatique

Pour le calcul des dérivées successives de $\tau$, nous pourrions dériver symboliquement, $n$ fois de suite, l'expression qui définit $\tau$ pour obtenir les dérivées successives $\tau^{(i)}$ $(i = 0, \ldots, n)$ formellement, puis les évaluer en $x_0$. Si les expressions sont stockées sous forme d'arbres, cette méthode est catastrophique car la taille de l'expression de $\tau^{(i)}$ croît en général très rapidement avec $i$. Par exemple, si on se contente d'appliquer successivement les règles usuelles de dérivation, sans faire de simplification formelle, la dérivée $i$-ème d'un simple produit $\tau = uv$ est une somme contenant $2^i$ termes.

Une amélioration significative consiste à représenter les expressions sous forme de graphes orientés acycliques (aussi appelés DAG, de l'anglais *directed acyclic graphs*) plutôt que sous forme

d'arbres : on évite ainsi de dupliquer les sous-expressions communes. De plus, cela permet de limiter le nombre de calculs lors de l'évaluation : la valeur d'une sous-expression n'est calculée qu'une seule fois, même si la sous-expression elle-même apparaît plusieurs fois.

Cependant, en l'occurrence, nous n'avons pas besoin de l'expression de la dérivée $i$-ème ; nous avons simplement besoin de connaître sa valeur en un point $x_0$. Or la dérivée de $\tau$ en $x_0$ ne dépend que de $\tau(x_0)$ et des dérivées des sous-expressions de $\tau$ évaluées en $x_0$. On peut donc se contenter de manipuler des nombres (les dérivées des sous-expressions de $\tau$ en $x_0$) au lieu de manipuler les expressions elles-mêmes. Cette idée, appelée différentiation automatique ou différentiation algorithmique, donne lieu à un domaine de recherche en soi.

Usuellement, la différentiation automatique est utilisée pour calculer la dérivée de fonctions pour lesquelles on n'a pas d'expression formelle. Typiquement, on a le code source d'un programme qui calcule une fonction numérique et on voudrait pouvoir calculer la dérivée de cette fonction numérique. La différentiation automatique est un processus qui permet de générer le code source d'un programme qui calcule la dérivée de la fonction numérique, à partir du code source de la fonction elle-même. La plupart du temps, le programme en question a de nombreuses variables en entrée et en sortie ; la difficulté principale consiste à savoir comment calculer efficacement la dérivée partielle que l'on souhaite et uniquement celle-ci. Notre cas est légèrement différent puisque :

— nous n'avons pas un programme mais l'expression d'une fonction numérique $\tau$ ;
— notre fonction est univariée et à valeurs réelles ;
— nous ne voulons pas calculer la dérivée première mais les $n$ premières dérivées ;
— nous ne souhaitons pas générer du code, mais effectuer le calcul à la volée, à partir de l'expression de $\tau$.

Cependant la nature du problème est la même et des techniques de type *différentiation automatique* existent pour calculer automatiquement et efficacement les dérivées successives d'une fonction en un point. Moore les explique déjà dans son livre [Moo79] en 1979. On retrouve la description des mêmes techniques dans des livres et articles récents [BS97, Gri00].

Puisqu'on veut calculer les coefficients de Taylor de $\tau$, il nous faut en fait la dérivée $i$-ème divisée par $i!$. Cette division peut être intégrée directement dans le calcul et nous allons, en fait, décrire des algorithmes pour calculer automatiquement $\tau^{(i)}(x_0)/i!$.

À toute fonction $\tau$, nous associons un tableau $[\tau_0, \ldots, \tau_n]$ tel que $\tau_i = \tau^{(i)}(x_0)/i!$. Notre but est donc exactement de calculer ce tableau. Vu sous un autre angle, ce tableau représente le développement (tronqué à l'ordre $n$) de la série de Taylor de $\tau$ en $x_0$. Il existe des algorithmes rapides pour calculer le produit, l'inverse, la composition, etc. de séries tronquées [BK75, BK78] ; cependant, nous nous contenterons dans la suite de décrire les algorithmes classiques (qui sont suffisants pour notre usage, où $n$ ne dépasse pas la centaine, en général).

Pour calculer le tableau $[\tau_0, \ldots, \tau_n]$, nous procédons par récurrence sur la structure de l'expression de $\tau$ :

— si $\tau$ est une constante $c$ : $\tau_0 = c$ et pour $i \geq 1$, $\tau_i = 0$ ;
— si $\tau$ est la variable, $\tau_0 = x_0$, $\tau_1 = 1$, et pour $i \geq 2$, $\tau_i = 0$ ;
— si $\tau$ est une addition : $\tau = u + v$ : puisque $\tau^{(i)} = u^{(i)} + v^{(i)}$, on a aussi $\tau_i = u_i + v_i$ pour tout $i$ ;
— si $\tau$ est une multiplication : $\tau = uv$ : la formule de Leibniz donne

$$\tau^{(i)} = \sum_{k=0}^{i} \binom{i}{k} u^{(k)} v^{(i-k)}.$$

Il suit que

$$\frac{\tau^{(i)}}{i!} = \sum_{k=0}^{i} \frac{u^{(k)}}{k!} \cdot \frac{v^{(i-k)}}{(i-k)!}.$$

Donc on a $\tau_i = \sum_{k=0}^{i} u_k v_{i-k}$ pour tout $i$. Ceci permet, étant donné le tableau des $u_i$ et des $v_i$ de calculer le tableau des $\tau_i$. L'algorithme correspondant `ADMult` est récapitulé en figure « algorithme 3.1 » ;

— si $\tau$ est la composée d'une fonction de base $u$ (c'est-à-dire $u(x) = x^\alpha$, $\alpha^x$, $\exp(x)$, $\sin(x)$, $\arccos(x)$, $\mathrm{erf}(x)$, etc.) et d'une autre fonction $v$, on peut utiliser au choix un algorithme *ad hoc* ou un algorithme générique. Nous allons à présent étudier ces deux éventualités.

```
1 Algorithme : ADMult
  Input : [u0, ..., un], [v0, ..., vn] ;        /* ui = u^(i)(x0)/i! et vi = v^(i)(x0)/i! */
  Output : [w0, ..., wn] ;                      /* wi = (uv)^(i)(x0)/i! */
2 for i ← 0 to n do
3 |   s ← 0 ;
4 |   for k ← 0 to i do
5 |   |   s ← s + uk * v(i-k) ;
6 |   end
7 |   wi ← s ;
8 end
9 return [w0, ..., wn] ;
```

**Algorithme 3.1 :** Différentiation automatique d'un produit

On pose $[v_0, \ldots, v_n]$ le tableau correspondant à $v$ au point $x_0$, et $[u_0, \ldots, u_n]$ le tableau correspondant à $u$ au point $v(x_0)$.

On a $\tau_0 = u_0$ et, pour tout $i \geq 0$,

$$\frac{(u \circ v)^{(i+1)}}{(i+1)!} = \frac{1}{i+1} \cdot \frac{\Big((u \circ v)'\Big)^{(i)}}{i!} = \frac{1}{i+1} \cdot \frac{\Big(v' \cdot (u' \circ v)\Big)^{(i)}}{i!}.$$

Posons $\sigma = u' \circ v$ et $\widetilde{v} = v'$. Nous avons alors

$$\tau_{i+1} = \frac{1}{i+1} \sum_{k=0}^{i} \widetilde{v}_k \, \sigma_{i-k}.$$

Or, $v_{k+1} = v^{(k+1)}/(k+1)! = (v')^{(k)}/(k+1)! = \widetilde{v}^{(k)}/(k+1)! = \widetilde{v}_k/(k+1)$. On en déduit donc l'équation finale suivante :

$$\tau_{i+1} = \frac{1}{i+1} \sum_{k=0}^{i} (k+1) v_{k+1} \, \sigma_{i-k}. \tag{3.3}$$

Ceci permet d'écrire un algorithme générique récursif pour la composée de deux fonctions : un appel récursif permet d'obtenir le tableau $[\sigma_0, \ldots, \sigma_{n-1}]$ correspondant à la fonction $\sigma$, puis les $\tau_i$ sont simplement calculés par une somme. Notons que l'algorithme ne manipule pas les expressions des fonctions $\tau = u \circ v$ et $\sigma = u' \circ v$, mais uniquement des tableaux de valeurs : les tableaux $[u_0, \ldots, u_n]$ et $[v_0, \ldots, v_n]$ pour l'appel principal, et les tableaux $[u_1, \ldots, u_n]$ et $[v_0, \ldots, v_{n-1}]$ pour l'appel récursif. L'algorithme ainsi obtenu est récapitulé en figure « algorithme 3.2 ».

L'algorithme `ADComp` est générique : il permet de composer n'importe quelles fonctions $u$ et $v$. Évidemment, il faut préalablement calculer les tableaux $[u_0, \ldots, u_n]$ et $[v_0, \ldots, v_n]$. Les $u_i$ s'obtiennent simplement en calculant les dérivées successives de $u$ ; puisque $u$ est une fonction de base (exp, arccos, etc.), on connaît généralement l'expression analytique des dérivées successives. Lorsque

```
1  Algorithme : ADComp
   Input : [u0, ..., un], [v0, ..., vn] ;        /* ui = (u^(i) ∘ v)(x0)/i! et vi = v^(i)(x0)/i! */
   Output : [w0, ..., wn] ;                      /* wi = (u ∘ v)^(i)(x0)/i! */
2  if n = 0 then
3  |  return [u0] ;
4  else
5  |  r ← ADComp([1 * u1, ..., n * un], [v0, ..., v(n-1)]) ;
6  |  s ← ADMult(r, [1 * v1, ..., n * vn]) ;
7  |  for i ← 1 to n do
8  |  |  wi ← si/i ;
9  |  end
10 |  return [u0, w1, ..., wn] ;
11 end
```

**Algorithme 3.2 :** Différentiation automatique d'une composée

ce n'est pas le cas (par exemple pour arccos) on sait du moins les calculer par une récurrence simple. Dans le cas de arccos par exemple, on sait que la dérivée $n$-ième est de la forme

$$\frac{p_n(x)}{(1+x^2)^{(2n-1)/2}} \quad \text{où } p_n \text{ vérifie la récurrence } p_{n+1} = (1+x^2)p_n' - (2n-1)\,x\,p_n.$$

Le tableau des $v_i$ est obtenu en appelant récursivement la procédure de différentiation automatique sur l'expression $v$.

On peut adopter une approche plus spécifique en écrivant une procédure de composition pour chaque fonction de base $u$. Prenons par exemple le cas de $u = \exp$. Puisque $u' = u$, l'équation (3.3) se simplifie :

$$\text{lorsque } \tau = \exp(v), \qquad \tau_{i+1} = \frac{1}{i+1}\sum_{k=0}^{i}(k+1)v_{k+1}.$$

Le tableau $[\tau_0, \ldots, \tau_n]$ se calcule donc séquentiellement, chaque terme étant obtenu comme combinaison des précédents.

De même, si $u$ est l'inverse d'une fonction qu'on sait facilement différencier, on peut tirer partie de cette propriété : prenons l'exemple $u = \ln$. Alors $\exp(\tau) = v$, donc en utilisant la formule précédente on a

$$v_{i+1} = \frac{1}{i+1}\sum_{k=0}^{i}(k+1)\tau_{k+1}v_{i-k}.$$

On en déduit une formulation de $\tau_{i+1}$ en fonction des $\tau_k$ précédents et des $v_k$ :

$$\text{lorsque } \tau = \ln(v), \qquad \tau_{i+1} = \frac{1}{(i+1)v_0}\left((i+1)v_{i+1} - \sum_{k=1}^{i} k\,\tau_k\,v_{i+1-k}\right).$$

On peut ainsi écrire des formules spécifiques pour la plupart des fonctions usuelles [Moo79, BS97, Gri00]. L'avantage de ces formules est qu'elles permettent de calculer le tableau $[\tau_0, \ldots, \tau_n]$ à partir de tableaux $[u_0, \ldots, u_n]$ et $[v_0, \ldots, v_n]$ en $\mathcal{O}(n^2)$ opérations, alors que l'algorithme générique `ADComp` nécessite $\mathcal{O}(n^3)$ opérations. L'algorithme `ADComp` reste utile pour toutes les fonctions $u$ pour lesquelles il n'est pas possible d'écrire une formule spécifique. En outre, nous avons observé que l'algorithme `ADComp` faisait apparaître moins de décorrélations lorsqu'on utilise la différentiation automatique avec des intervalles.

L'algorithme AD complet qui calcule le tableau $[\tau_0, \ldots, \tau_n]$ à partir de l'expression de $\tau$ est résumé en figure « algorithme 3.3 ».

### 3.4.2 Calcul explicite du polynôme $T$

L'outil de différentiation automatique nous permet donc de calculer facilement les $n$ premières dérivées de n'importe quelle fonction (divisée par la factorielle correspondante). Cependant, puisque nous voulons certifier les calculs, nous ne pouvons utiliser la différentiation automatique telle quelle.

Les algorithmes de différentiation automatique peuvent parfaitement être utilisés avec l'arithmétique d'intervalle : si $x_0$ est remplacé par un intervalle $\boldsymbol{x}$, et si on applique exactement les mêmes algorithmes en utilisant l'arithmétique d'intervalle, le résultat sera un intervalle $\boldsymbol{\tau_i^x}$ englobant l'image exacte $\tau^{(i)}(\boldsymbol{x})/i!$.

En appliquant la différentiation automatique à un intervalle-point $\boldsymbol{x} = [x_0, x_0]$ et en utilisant de l'arithmétique d'intervalle multiprécision, on obtient un encadrement arbitrairement fin de la valeur exacte $\tau^{(i)}(x_0)/i!$. Ceci va nous permettre de calculer effectivement et de façon certifiée un polynôme $T$ aussi proche qu'on le désire du polynôme de Taylor exact. En général, le polynôme $T$ n'est pas exactement égal au polynôme de Taylor (car les coefficients de celui-ci ne sont pas exactement représentable en virgule flottante) ; nous devons donc borner l'erreur entre le polynôme de Taylor exact et le polynôme $T$ que nous allons calculer. Par ailleurs, nous devons borner rigoureusement le reste $\Delta_n(x, \xi)$ de la formule de Taylor (équation (3.2), page 156).

a) Il est aisé de borner le terme $(x-x_0)^i$ lorsque $x, x_0 \in [a, b]$ et $i \in [\![0, n]\!]$. Soit $\boldsymbol{x_i}$ un intervalle englobant ce terme : $\forall x, x_0 \in [a, b]\,, \forall i \in [\![0, n]\!]\,, (x - x_0)^i \in \boldsymbol{x_i}$.

b) Pour borner $\Delta_n(x, \xi)$, il suffit de calculer

$$\boldsymbol{\tau_n^{[a,b]}} \ni \frac{\tau^{(n)}(\xi)}{n!}, \quad \text{où } \xi \in [a, b]$$

en utilisant AD($\tau$,$n$,$[a, b]$). On pose alors $\boldsymbol{\Delta} = \boldsymbol{\tau_n^{[a,b]}} \cdot \boldsymbol{x_n}$, de sorte que $\Delta_n(x, \xi) \in \boldsymbol{\Delta}$.

c) Les coefficients de Taylor $c_i$ n'étant pas exactement représentables, il nous faut, en fait, calculer de petits intervalles englobant la valeur exacte de chaque $c_i$ : nous obtenons de tels intervalles $\boldsymbol{c_i} \ni c_i$ en utilisant AD($\tau$,$n-1$,$[x_0, x_0]$). Puisqu'il s'agit d'un intervalle-point, l'encadrement peut être rendu arbitrairement fin en augmentant la précision des calculs.

d) Enfin, on calcule effectivement le polynôme d'approximation ; on choisit un point $t_i$ dans $\boldsymbol{c_i}$ (proche du milieu) et on définit notre polynôme d'approximation $T$ par

$$T(x) = \sum_{i=0}^{n-1} t_i (x - x_0)^i.$$

La différence entre le vrai polynôme de Taylor et celui que nous avons calculé est facile à borner en utilisant l'arithmétique d'intervalle : on a $c_i \in \boldsymbol{c_i}$ et donc, $(c_i - t_i) \in [\underline{\boldsymbol{c_i}} - t_i, \overline{\boldsymbol{c_i}} - t_i]$, d'où finalement

$$\sum_{i=0}^{n-1} (c_i - t_i)(x - x_0)^i \in \sum_{i=0}^{n-1} [\underline{\boldsymbol{c_i}} - t_i, \overline{\boldsymbol{c_i}} - t_i] \cdot \boldsymbol{x_i} = \boldsymbol{\delta}.$$

L'erreur finale totale entre $T$ et $\tau$ est donc comprise dans l'intervalle $\boldsymbol{\delta} + \boldsymbol{\Delta}$ :

$$\forall x \in [a, b]\,, \tau(x) - T(x) \in \boldsymbol{\delta} + \boldsymbol{\Delta}.$$

Soit $\theta \in \mathbb{R}^+$ le plus petit réel tel que $\boldsymbol{\delta} + \boldsymbol{\Delta} \subseteq [-\theta, \theta]$. Si $\theta$ est suffisamment petit, on peut utiliser le polynôme $T$. Si ce n'est pas le cas, il faut soit augmenter le degré de $T$, soit diviser l'intervalle $[a, b]$ en deux et calculer une approximation sur chaque demi-intervalle.

```
1  Algorithme : AD
   Input : τ, n, x_0 ; /* τ est une fonction donnée sous la forme d'une expression */
   Output : [w0, ..., wn] ;                                  /* wi = τ^(i)(x_0)/i! */
2  switch τ do
3      case une constante c
4          w0 ← c ;
5          for i ← 1 to n do
6              wi ← 0 ;
7          end
8      end
9      case la variable x
10         w0 ← x_0 ;
11         w1 ← 1 ;
12         for i ← 2 to n do
13             wi ← 0 ;
14         end
15     end
16     case u + v
17         [u0, ..., un] ← AD(u) ;
18         [v0, ..., vn] ← AD(v) ;
19         for i ← 0 to n do
20             wi ← ui + vi ;
21         end
22     end
23     case uv
24         [u0, ..., un] ← AD(u) ;
25         [v0, ..., vn] ← AD(v) ;
26         w ← ADMult([u0, ..., un], [v0, ..., vn]) ;
27     end
28     case exp(v)
29         [v0, ..., vn] ← AD(v) ;
30         w0 ← exp(v0) ;
31         for i ← 1 to n do
32             s ← 0 ;
33             for k ← 0 to i do
34                 s ← s + (k + 1) * v(k+1) ;
35             end
36             wi ← s/(i + 1) ;
37         end
38     end
39     case ···
40         ··· ;
41     end
42 end
43 return [w0, ..., wn] ;
```

**Algorithme 3.3 :** Différentiation automatique d'une expression $\tau$

### 3.4.3 Autres méthodes pouvant mener à un reste plus petit

Comme nous l'avons vu, il est relativement aisé de calculer un polynôme de Taylor approchant $\tau$ et il est facile de borner automatiquement le reste en utilisant la formulation de Lagrange. Cependant, le reste peut être assez largement surestimé par l'arithmétique d'intervalle ; de plus (indépendamment de cette surestimation) le polynôme de Taylor n'est pas une très bonne approximation lorsque l'intervalle $[a, b]$ grandit. Dans ce cas, il faut utiliser un polynôme de degré très élevé ou bien couper l'intervalle en sous-intervalles plus petits sur lesquels les polynômes de Taylor seront de qualité suffisante.

Un certain nombre de méthodes sont envisageables pour obtenir de meilleurs polynômes ou un reste moins surestimé. Nous n'avons pas encore eu le temps d'implanter les algorithmes correspondants pour faire une étude comparative. Les diverses méthodes envisagées sont les suivantes :

— utilisation d'un polynôme interpolateur : il est possible de borner explicitement l'erreur entre un polynôme interpolateur et la fonction qu'il interpole [Che82, Chap. 3]. La borne fait intervenir les points d'interpolation et la dérivée $n$-ième de la fonction interpolée $\tau$. Cette méthode reste donc sujette au problème de surestimation due à l'utilisation de l'arithmétique d'intervalle pour calculer l'encadrement de $\tau^{(n)}([a, b])$. Cependant, si les points d'interpolation sont bien choisis, le polynôme obtenu est souvent de bien meilleure qualité que le polynôme de Taylor. On peut prendre comme points les zéros du polynôme de Tchebychev de degré $n$ (voir équation (2.9) en page 127) : ils fournissent en règle générale une approximation d'excellente qualité, assez proche de ce que donne le polynôme de meilleure approximation ;

— utilisation des modèles de Taylor : les modèles de Taylor [MB03, Neu03] permettent de calculer simultanément un polynôme d'approximation et un encadrement du reste sous la forme d'un intervalle (contrairement à ce que nous avons développé plus haut, où le polynôme et le reste étaient calculés séparément). Comme leur nom l'indique, les modèles de Taylor ont été conçus, à l'origine, pour calculer un polynôme de Taylor. Pour ce que nous avons pu en observer, le reste est borné plus finement par les modèles de Taylor que lorsqu'on utilise le reste de Lagrange et l'arithmétique d'intervalle. La technique des modèles de Taylor peut également être utilisée pour calculer des polynômes d'approximation de meilleure qualité que le polynôme de Taylor ;

— si la fonction $\tau$ est analytique sur un voisinage complexe de l'intervalle $[a, b]$ (hypothèse couramment vérifiée en pratique), il est possible d'utiliser une technique de série majorante. On s'appuie alors sur le fait que le reste d'un polynôme de Taylor s'exprime comme la queue d'une série :

$$\tau(x) - T(x) = \sum_{i=n}^{+\infty} \frac{\tau^{(i)}(x_0)}{i!} (x - x_0)^i = \sum_{i=n}^{+\infty} c_i (x - x_0)^i.$$

Pour borner cette série, on majore la suite $(|c_i|)_{i \geq n}$ par une suite $(\gamma_i)_{i \geq n}$ (par exemple une suite géométrique). On a alors

$$\forall x \in [a, b]\,,\ |\tau(x) - T(x)| \leq \underbrace{\sum_{i=n}^{+\infty} \gamma_i |x - x_0|^i}_{\text{calculable explicitement}}\,.$$

La formule de Cauchy, par exemple, permet d'obtenir une telle suite $(\gamma_i)_{i \geq n}$ et peut fournir une borne de très bonne qualité [Neh01, Neh03] ;

— d'autres méthodes récentes semblent permettre de calculer des polynômes d'approximation et des restes de bonne qualité automatiquement et efficacement [vdH07]. Nous n'avons pas eu le temps de les étudier précisément.

## 3.5 Localisation des zéros d'une fonction

### 3.5.1 Cas des polynômes

Nous abordons à présent le problème de la localisation fine et certifiée des zéros d'un polynôme. Étant donné un polynôme univarié $T$ et un intervalle $[a, b]$, nous voulons calculer une liste d'intervalles très fins contenant (avec certitude) *toutes* les racines $z$ de $T$ telles que $z \in [a, b]$. Il existe de nombreuses méthodes pour localiser ainsi les racines d'un polynôme (citons par exemple l'algorithme de Newton par intervalle ou la méthode d'exclusion [DY93]) ; nous nous sommes plus particulièrement intéressés à deux grandes familles de méthodes, que nous décrivons maintenant.

Une première famille de méthodes s'appuie sur la *règle des signes de Descartes*. Cette règle permet d'obtenir un majorant du nombre de racines strictement positives de $T$ (comptées avec multiplicité). Il suffit pour cela de compter le nombre de changements de signes dans la liste ordonnée des coefficients de $T$ (en ne tenant pas compte des coefficients nuls). Pour se ramener à l'intervalle $[a, b]$, il faut faire un ensemble de transformations sur le polynôme $T$ (translation, homothétie, renversement). Puisque la règle des signes ne donne qu'un majorant du nombre de racines, il faut l'utiliser conjointement avec une stratégie dichotomique pour réduire la taille de l'intervalle et le nombre de racines qu'il contient. Si $T$ est sans facteur carré, le processus s'arrête forcément lorsque la taille de l'intervalle devient assez petite [RZ04]. Il est possible d'éliminer *a priori* les facteurs carrés de $T$ en calculant le pgcd de $T$ et de sa dérivée $T'$.

Une deuxième famille de méthodes s'appuie sur le calcul de la *suite de Sturm* associée au polynôme $T$ [Stu35, Bro96] : il s'agit de la suite $T_0, T_1, T_2, \ldots$ où

$$T_0 = T,$$
$$T_1 = T',$$

et pour tout $i \geq 0$, $T_{i+2}$ est l'opposé du reste de la division euclidienne de $T_i$ par $T_{i+1}$.

Si $n_a$ désigne le nombre de changements de signes dans la suite des $T_i(a)$ (là encore, on ne tient pas compte des valeurs nulles) et si $n_b$ désigne le nombre de changements de signes dans la suite des $T_i(b)$, le nombre exact de racines de $T$ dans l'intervalle $[a, b]$ est donné par $n_a - n_b$. Une fois qu'on connaît le nombre de racines dans l'intervalle, un algorithme de recherche numérique des zéros peut être employé pour les localiser. Pour chaque zéro $z$, il suffit alors de se décaler légèrement sur la gauche et sur la droite : $\underline{z} \leq z \leq \overline{z}$. On vérifie que $T(\underline{z})$ et $T(\overline{z})$ sont de signes opposés : pour être sûr du signe de $T(\underline{z})$ (respectivement $T(\overline{z})$), on évalue $T$ par arithmétique d'intervalle sur l'intervalle-point $[\underline{z}, \underline{z}]$ (respectivement $[\overline{z}, \overline{z}]$). Quitte à augmenter la précision, on obtient comme résultat un intervalle suffisamment fin autour de la valeur exacte $T(\underline{z})$ pour décider du signe.

Le calcul de la suite de Sturm est connu pour poser des problèmes numériques [Bro96] et pour avoir une complexité moins intéressante que les algorithmes fondés sur la règle des signes de Descartes [RZ01]. En particulier, F. Rouillier et P. Zimmermann proposent dans [RZ04] un algorithme d'isolation des zéros d'un polynôme qui s'appuie sur la règle de Descartes et l'arithmétique d'intervalle. Dans leur algorithme, les coefficients des polynômes sont remplacés par des petits intervalles représentant leur valeur exacte. Cette technique permet de réduire la taille en bits des données manipulées et donc le coût de chaque opération arithmétique.

Dans notre implémentation actuelle, nous utilisons la méthode de Sturm : ainsi, nous n'avons pas à vérifier au préalable que $T$ est sans facteur carré. De la même façon que Rouillier et Zimmermann, les coefficients des polynômes que nous manipulons sont des petits intervalles.

### 3.5.2 Des polynômes aux fonctions

Nous en arrivons maintenant au problème qui nous intéresse réellement : localiser finement les zéros de la dérivée $\varepsilon'$ d'une fonction d'erreur. Dans le cas de l'erreur relative, nous avons vu qu'il

était préférable de chercher les zéros de la fonction $p'f - f'p$. Ce sont tous des zéros de $\varepsilon'$ et on évite ainsi la division par $f^2$ qui pourrait poser problème lors du calcul de la borne de Lagrange par arithmétique d'intervalle.

On définit donc la fonction intermédiaire $\tau$ suivante :

$$\begin{aligned} \tau &= p' - f' && \text{si l'on considère l'erreur absolue ;} \\ \tau &= p'f - f'p && \text{si l'on considère l'erreur relative.} \end{aligned}$$

Nous voulons calculer une liste $\mathcal{Z}$ contenant des intervalles $\boldsymbol{z_i}$ arbitrairement fins, et telle que tout zéro de $\tau$ se trouve dans l'un au moins des $\boldsymbol{z_i}$. Il n'est pas gênant pour notre algorithme que les intervalles $\boldsymbol{z_i}$ contiennent plusieurs zéros de $\tau$, du moment que les $\boldsymbol{z_i}$ soient de taille suffisamment petite. De même, il n'est pas gênant que quelques $\boldsymbol{z_i}$ ne contiennent en réalité aucun zéro de $\tau$. Ce qui est important, c'est d'être sûr qu'aucun zéro de $\tau$ n'a été oublié.

La fonction $\tau$ est remplacée par un polynôme $T$ tel que $\|T - \tau\|_\infty \leq \theta$. Nous devons maintenant montrer comment obtenir la liste $\mathcal{Z}$ à partir de $T$. Pour cela, nous nous appuyons sur le fait que, si $z$ est tel que $\tau(z) = 0$, alors $|T(z)| \leq \theta$. Il suffit donc de trouver les points $z \in [a, b]$ pour lesquels $T(z)$ se situe dans la bande délimitée par $-\theta$ et $\theta$ (voir figure 3.2). Plus formellement, nous recherchons les points $z$ pour lesquels simultanément $T(z) \leq \theta$ et $T(z) \geq -\theta$.

Supposons un instant que l'on soit capable de calculer une liste $\mathcal{L}_u$ d'intervalles représentant les points où $T$ est inférieur à $\theta$. De la même façon, soit $\mathcal{L}_\ell$ une liste d'intervalles représentant les points où $T$ est supérieur à $-\theta$. Il est clair qu'on peut calculer, à partir de ces deux listes, des intervalles $\boldsymbol{z_i}$ couvrant l'ensemble des points où $T$ est dans la bande délimitée par $-\theta$ et $\theta$. Précisément, il suffit de prendre l'intersection des intervalles contenus dans $\mathcal{L}_u$ et des intervalles contenus dans $\mathcal{L}_\ell$ (voir figure 3.2).

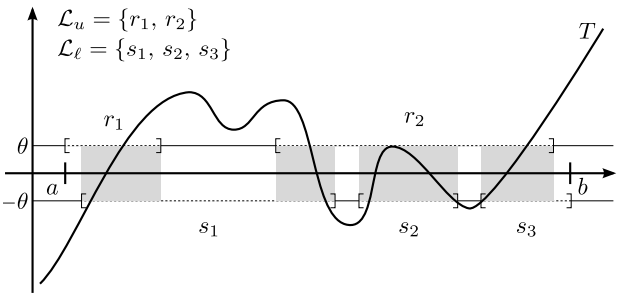



**Figure 3.2 :** Comment calculer $\mathcal{Z}$
(les intervalles $\boldsymbol{r_i}$ et $\boldsymbol{s_i}$ sont délibérément agrandis à des fins d'illustration)

Il nous reste à voir comment calculer la liste $\mathcal{L}_u$ (et de même la liste $\mathcal{L}_\ell$). $\mathcal{L}_u$ représente l'ensemble des points $x$ où $T(x) \leq \theta$, c'est-à-dire $T(x) - \theta \leq 0$. Or $T - \theta$ est un polynôme et n'a donc qu'un nombre fini de racines dans l'intervalle $[a, b]$. De plus, le lieu des points où $T - \theta \leq 0$ est délimité par les zéros de $T - \theta$ (car $T - \theta$ est continue). Il suffit donc de trouver les points où $T - \theta$ s'annule et de regarder le signe à gauche et à droite de chaque zéro pour savoir où $T - \theta$ est positif et où il est négatif.

Puisque nous voulons un résultat certifié, nous devons être rigoureux. Il faut donc utiliser une technique d'isolation des zéros d'un polynôme décrite dans la section précédente. Celle-ci nous fournit une liste $\mathcal{Z}_u = \{\boldsymbol{z_1}, \ldots, \boldsymbol{z_k}\}$ des racines de $T - \theta$ (les $\boldsymbol{z_i}$ sont des intervalles fins encadrant les racines).

Pour chaque $\boldsymbol{z} \in \mathcal{Z}_u$, nous déterminons le signe de $T - \theta$ à gauche et à droite de $\boldsymbol{z}$ : pour cela, on choisit un point $x$ à gauche (respectivement à droite) de $\boldsymbol{z}$ et on évalue $(T - \theta)$ par arithmétique d'intervalle sur l'intervalle-point $[x, x]$. Évidemment, il faut être sûr que $T - \theta$ n'a pas changé de

signe entre $x$ et $\boldsymbol{z}$. Puisque la racine précédente de $T-\theta$ est connue (elle est dans l'intervalle $\boldsymbol{z'}$ précédant $\boldsymbol{z}$ dans la liste $\mathcal{Z}_u$), il suffit de prendre $x$ n'importe où entre $\boldsymbol{z'}$ et $\boldsymbol{z}$.

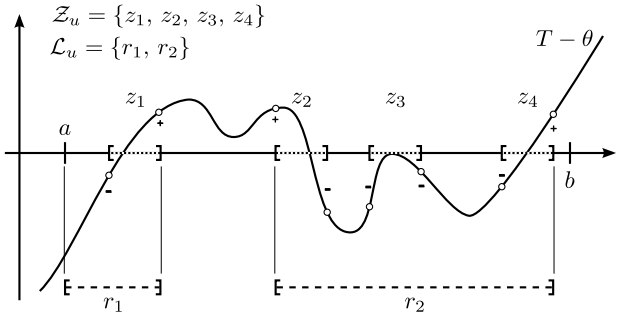



**Figure 3.3 :** Comment calculer $\mathcal{L}_u$

La liste $\mathcal{L}_u$ est alors facile à obtenir. Si $\boldsymbol{z_i}$ est un zéro caractérisé par les signes $(+, -)$, cela signifie que $T-\theta$ devient négatif après $\boldsymbol{z_i}$. Il redevient positif avec le premier $\boldsymbol{z_j}$ suivant caractérisé par les signes $(-, +)$. Il faut donc ajouter $\left[\underline{\boldsymbol{z_i}}, \overline{\boldsymbol{z_j}}\right]$ à la liste $\mathcal{L}_u$ (où $\underline{\boldsymbol{z_i}}$ est la borne inférieure de $\boldsymbol{z_i}$ et $\overline{\boldsymbol{z_j}}$ est la borne supérieure de $\boldsymbol{z_j}$). Les premier et dernier zéros sont des cas particuliers traités à part. La construction de $\mathcal{L}_u$ est illustrée par la figure 3.3.

## 3.6 Résultats expérimentaux

Nous avons implémenté les deux algorithmes de calcul de norme sup présentés dans ce chapitre. Le premier est inclus dans Sollya sous la forme de la commande `infnorm`. Cet algorithme ayant montré ses limites, il devrait bientôt être remplacé par le second algorithme qui permet d'obtenir plus rapidement des résultats plus fins. Le deuxième algorithme a été implanté sous la forme d'un prototype qu'on peut utiliser dans Sollya, moyennant le chargement de quelques fonctions externes. En particulier, nous avons implémenté l'algorithme de différentiation automatique, l'algorithme de Sturm pour dénombrer les racines d'un polynôme contenues dans un intervalle, ainsi que l'algorithme de Newton par intervalle qui est une façon alternative d'encadrer les zéros d'une fonction sur un intervalle ; nous avons écrit des procédures en langage Sollya pour calculer de façon certifiée le polynôme de Taylor et son reste, ainsi que des procédures pour encadrer les zéros d'une fonction numérique quelconque.

| Exemple n° | $f$ | $[a, b]$ | $\deg(p)$ | mode | $\deg(T)$ | qualité | temps |
|---|---|---|---|---|---|---|---|
| 1 | $\exp(x)-1$ | $[-0.25, 0.25]$ | 5 | relative | 11 | 37.6 | 0.4 s |
| 2 | $\log_2(1+x)$ | $[-2^{-9}, 2^{-9}]$ | 7 | relative | 23 | 83.3 | 2.2 s |
| $3^1$ | $\arcsin(x+m)$ | $[a_3, b_3]$ | 22 | relative | 37 | 15.9 | 5.1 s |
| 4 | $\cos(x)$ | $[-0.5, 0.25]$ | 15 | relative | 28 | 19.5 | 2.2 s |
| 5 | $\exp(x)$ | $[-0.125, 0.125]$ | 25 | relative | 41 | 42.3 | 7.8 s |
| 6 | $\sin(x)$ | $[-0.5, 0.5]$ | 9 | absolue | 14 | 21.5 | 0.5 s |
| 7 | $\exp(\cos(x)^2+1)$ | $[1, 2]$ | 15 | relative | 60 | 25.5 | 11.0 s |
| 8 | $\tan(x)$ | $[0.25, 0.5]$ | 10 | relative | 21 | 26.0 | 1.1 s |
| 9 | $x^{2.5}$ | $[1, 2]$ | 7 | relative | 26 | 15.5 | 1.4 s |

**Figure 3.4 :** Résultats du second algorithme sur divers exemples

Le tableau 3.4 présente neuf exemples recouvrant des situations variées. Les expériences ont été faites sur un ordinateur équipé d'un processeur Intel Pentium D à 3.00 GHz avec 2 Go de RAM, tournant sous GNU/Linux et avec le compilateur gcc.

Pour chaque exemple, nous donnons le temps nécessaire au calcul de l'encadrement de la norme sup ainsi que la qualité de l'encadrement obtenu. Si l'algorithme calcule un encadrement $[\ell, u]$, la qualité de l'encadrement $-\log_2(u/\ell - 1)$ indique approximativement la précision (en bits) obtenue en considérant $u$ comme approximation de la valeur exacte $\|\varepsilon\|_\infty$. Pour le développement de `libms` telle que `CRlibm` par exemple, il n'est généralement pas nécessaire de connaître plus que quelques bits de la norme sup. Avoir 15 bits de précision est déjà beaucoup.

Les deux premiers exemples sont ceux présentées dans notre article [CL07] et permettent une comparaison entre le premier algorithme et le second. Le second exemple, lorsqu'il est traité avec le premier algorithme, prend environ 120 fois plus de temps qu'avec le second algorithme.

Le troisième exemple est un polynôme directement tiré du code source de `CRlibm`. Il s'agit d'un exemple typique du genre d'erreurs d'approximation manipulées par les développeurs de `libms`. Le degré de $p$ est 22, ce qui est déjà assez élevé. Notre algorithme traite l'exemple en 5 secondes.

Dans les exemples 4 à 9, le polynôme $p$ est le polynôme de meilleure approximation, obtenu par l'algorithme de Rémez. Ces exemples mettent en jeu des fonctions plus ou moins compliquées sur des intervalles plus ou moins larges. En particulier les exemples 7 et 9 doivent être considérés comme relativement durs pour notre algorithme puisque l'intervalle de définition est de taille 1 : il s'agit d'une taille assez large lorsqu'on utilise les polynômes de Taylor, puisque le terme $(b-a)^n$ dans le reste de Lagrange ne contribue pas à diminuer le reste. Cela nous force donc à utiliser un polynôme de Taylor de degré assez élevé. Tous les exemples présentés sont traités en moins de 15 secondes. Il s'agit d'un temps de calcul très raisonnable pour une procédure qui, *a priori*, n'est appelée qu'une fois lors du développement d'une fonction numérique dans une `libm`.

## 3.7 Conclusion et perspectives

Lorsqu'on prétend fournir une bibliothèque mathématique correctement arrondie, il est nécessaire de donner des garanties sérieuses sur la qualité de l'implémentation. Cela implique, en particulier, de borner rigoureusement l'erreur d'approximation $\varepsilon = p/f - 1$ entre la fonction $f$ et le polynôme $p$ utilisé pour son évaluation. En d'autres termes, il faut calculer un majorant fin de $\|\varepsilon\|_\infty$.

Il existe un certain nombre d'algorithmes d'optimisation globale certifiée. Ils reposent sur une utilisation plus ou moins fine de l'arithmétique d'intervalle ; une stratégie de type *branch-and-bound* permet de réduire la taille des intervalles considérés, tout en ne gardant que ceux susceptibles de contenir les extrema globaux. Ces algorithmes sont complètement inefficaces sur les problèmes issus de l'approximation : l'arithmétique d'intervalle ignore la corrélation entre $p$ et $f$ et ne fournit aucune information, à moins que les intervalles considérés soient minuscules. En pratique, le temps de calcul et la mémoire nécessaires ne sont pas raisonnables.

Nous avons proposé une première approche pour résoudre ce problème, en améliorant l'arithmétique d'intervalle. Cela nous a permis de réduire la décorrélation et de traiter des cas de difficulté intermédiaire. Cependant, ce n'était pas suffisant pour aborder toutes les situations concrètes, telles que celles rencontrées dans le développement de `CRlibm`.

Notre deuxième approche s'attaque directement au problème de la décorrélation. La technique utilisée est classique : si une fonction $\tau$ présente un problème de décorrélation, on peut remplacer $\tau$ par un polynôme $T$ qui l'approche très bien. Le point important est que $T$ est choisi de telle sorte

1. Dans l'exemple n° 3 : $m = \dfrac{770422123864867}{2^{50}}$, $a_3 = -\dfrac{205674681606191}{2^{53}}$, $b_3 = \dfrac{205674681606835}{2^{53}}$.

que l'erreur $\tau - T$ soit facile à borner, même grossièrement. Dans notre implémentation, nous utilisons un polynôme de Taylor, calculé par différentiation automatique.

Notre méthode repose sur la recherche des zéros de la fonction $\varepsilon'$. Intuitivement, nous remplaçons $\varepsilon'$ par un polynôme $T$, en s'assurant que l'erreur $\varepsilon' - T$ ne dépasse pas un certain paramètre $\theta$. Puis nous recherchons les zéros de $T-\theta$ et $T+\theta$. Pour cela, nous utilisons une technique d'isolation certifiée des zéros de polynômes. Enfin, nous en déduisons des petits intervalles englobant de façon certaine les zéros de $\varepsilon'$. Ces intervalles sont suffisamment petits pour que l'arithmétique d'intervalle donne des résultats pertinents lorsqu'on évalue $\varepsilon$ sur eux.

Notre algorithme donne des résultats rigoureux et précis en un temps qui varie de quelques dizaines de secondes à quelques minutes. C'est très raisonnable pour un calcul certifié, qui n'est réalisé qu'une seule fois lors de l'implémentation d'une fonction dans une libm.

Nous envisageons plusieurs pistes d'amélioration : plutôt qu'un polynôme de Taylor, nous pourrions utiliser un polynôme d'interpolation aux point de Tchebychev. Ceci devrait permettre de réduire le degré du polynôme $T$ utilisé. Une autre possibilité serait de garder un polynôme de Taylor, mais en bornant le reste différemment, par exemple en utilisant la technique des *modèles de Taylor*, ou en utilisant des outils d'analyse complexe telle que la formule de Cauchy. Grâce à ces méthodes, le reste devrait être moins surestimé ; là encore, le bénéfice attendu est principalement une diminution du degré de $T$.

Pour isoler les zéros d'un polynôme, nous utilisons actuellement un algorithme fondé sur la suite de Sturm. Nous envisageons d'essayer d'autres méthodes comme celles fondées sur la règle des signes de Descartes, par exemple.

Notre algorithme souffre actuellement d'un petit défaut : il dépend du paramètre $\theta$ qui doit être choisi manuellement. Ce paramètre détermine la qualité de l'encadrement final de la norme sup. Cependant, le lien entre les deux n'est pas évident et, en pratique, on choisit $\theta$ suffisamment petit, en espérant obtenir au moins quelques bits corrects dans le résultat final. Si ce n'est pas le cas, il faut relancer l'algorithme avec une nouvelle valeur de $\theta$ inférieure à la précédente. À l'inverse, il arrive parfois que la valeur de $\theta$ initialement choisie soit bien plus petite que nécessaire : on obtient un résultat avec plusieurs dizaines de bits de précision qui sont complètement inutiles. Le temps de calcul aurait pu être bien plus faible si $\theta$ avait eu une valeur adaptée. Nous aimerions trouver un moyen de déterminer automatiquement une valeur appropriée de $\theta$ en fonction de la qualité finale désirée par l'utilisateur.

Enfin, une limitation actuelle de notre algorithme vient du fait que nous ne fournissons pas de preuve formelle. Une telle preuve est souhaitable, car elle permet d'offrir une garantie maximale : bien que notre algorithme ait été conçu rigoureusement, son implémentation pourrait contenir des bugs subtils. L'idéal serait de fournir non seulement le résultat du calcul, mais également une preuve que ce résultat est bien un encadrement de la valeur exacte. Cette preuve pourrait ensuite être vérifiée automatiquement par ordinateur. Pour atteindre cet objectif, il faudra voir dans quelle mesure les techniques d'isolation des zéros d'un polynôme peuvent être vérifiées formellement. En outre, il sera nécessaire de disposer d'une implémentation de l'arithmétique d'intervalle en précision arbitraire au sein de l'assistant de preuve formelle. Des travaux très prometteurs [Mel08] ont déjà été faits sur le sujet et nous pourrons envisager de les utiliser.

Chapitre 4

# Réalisation logicielle : Sollya, boîte à outils du numéricien

*Ce chapitre fait le point sur le logiciel Sollya. Nous revenons d'abord sur sa genèse ; ainsi nous mettons en lumière ce qui a constitué les forces et les faiblesses du développement de l'outil jusqu'à présent. Dans un second temps, nous analysons les perspectives d'évolution de Sollya et les questions de recherche associées.*

Nous avons présenté rapidement au début de ce manuscrit le logiciel Sollya. Le développement de ce logiciel a occupé une partie importante du temps de la présente thèse. Il a été le lieu naturel d'implantation des algorithmes décrits tout au long du manuscrit. Avec le logiciel Sollya, nous voulons, à terme, offrir au numéricien un outil intégré complet dans lequel il peut expérimenter rapidement et sans se soucier d'éventuels problèmes numériques. Revenons donc à présent sur ses forces et ses faiblesses ainsi que sur les axes de développement futurs.

## 4.1 Bref retour sur l'historique de Sollya

Sollya est né un soir, alors que Christoph Lauter et moi-même tentions d'expérimenter une idée d'algorithme pour le calcul d'un majorant de la norme sup d'une fonction. Nous avions besoin d'une interface minimale dans laquelle nous pourrions définir une fonction par une expression puis l'évaluer en précision arbitraire en virgule flottante ou par intervalle.

Nous n'avions pas clairement l'intention de développer un vrai logiciel, mais simplement un prototype rapide. Cette genèse n'est pas innocente car elle explique un certain nombre des choix qui sont à la base de Sollya.

Rapidement, l'outil devient le lieu naturel de prototypage des algorithmes que nous développons (Rémez, norme sup certifiée). Pour d'évidentes raisons pratiques, nous ajoutons au fur et à mesure les briques indispensables aux numériciens que nous sommes (différentiation formelle, développement de Taylor, mise sous forme de Horner, support des format `double`, double-`double`, triple-`double`, recherche de racines par itération de Newton, calcul approché rapide de la norme sup d'une fonction). Pour nous, Sollya est avant tout une sorte de calculette évoluée permettant d'utiliser directement `MPFR` et `MPFI` sans avoir à compiler un programme écrit en C à chaque fois.

Cependant, il y a une chose dont nous n'avions pas conscience : tôt ou tard, nous éprouverions le besoin d'ajouter à Sollya un langage de script. Il s'agit, je pense, d'un besoin qui apparaît naturellement avec tout logiciel qui se présente sous la forme d'un shell interactif et avec lequel on fait des expériences. Un jour ou l'autre, on désire tester l'égalité de deux valeurs, ou bien évaluer une expression en chaque zéro d'une fonction, c'est-à-dire faire une boucle. Nous avons donc ajouté un langage de script.

D'autres membres de l'équipe Arénaire ont commencé à utiliser le logiciel pour leurs propres expérimentations. Pour Christoph Lauter et moi-même, Sollya était l'endroit où implémenter les algorithmes que nous développions pendant nos thèses respectives. Ce faisant, nous rendions accessibles et librement utilisables les algorithmes que nous présentions dans nos articles. En outre, cela nous permettait d'inclure dans nos publications des exemples réalistes : ainsi, pour présenter un algorithme de norme sup certifiée, nous pouvions utiliser de véritables polynômes de meilleure approximation, obtenus par l'algorithme de Rémez. Nous pouvions en outre fournir des petits scripts qui permettent à tout un chacun de reproduire les exemples présentés dans un article.

À l'heure actuelle, et à notre connaissance, Sollya est utilisé par une dizaine de personnes, dans l'équipe Arénaire à Lyon bien sûr, mais aussi dans l'équipe DALI à Perpignan, à l'École polytechnique et par quelques ingénieurs de la société Intel.

## 4.2 Bilan de trois ans de développement

Sollya est né pour répondre rapidement à un besoin de prototypage. Il a donc les avantages, mais aussi les nombreux inconvénients, d'un logiciel développé suivant ce type d'approche.

Parmi les avantages, on compte la souplesse et la réactivité du développement. Lorsqu'il faut intégrer une fonctionnalité nouvelle, la décision est vite prise et la fonctionnalité rapidement programmée et utilisable. Autre avantage : nous avons une totale compréhension du fonctionnement de l'outil et donc une relative confiance dans ses résultats. Sollya n'est pas exempt de bugs bien sûr, mais lorsqu'un problème se présente, il est rapidement compris, isolé et corrigé.

Puisque nous n'étions que deux développeurs, les décisions étaient vite prises ; nous pouvions emmener le logiciel rapidement dans la direction que nous voulions, sans avoir à tenir compte de contraintes externes, de procédures lourdes ou de standards à respecter. Nous aurions pu choisir de développer les mêmes fonctionnalités au sein de logiciels libres déjà existants ; mais dans ce cas, nous aurions été contraints d'utiliser les structures de données déjà en place, de respecter les règles de codage propres à un projet déjà établi, de nous adapter à un logiciel dont les grands principes différaient de ce que nous souhaitions mettre en place. Nous aurions alors perdu en souplesse.

Cette souplesse de développement nous a permis de trouver en Sollya le logiciel dont nous avions exactement besoin pour nos expériences quotidiennes. Un exemple est donné par les développements récents de la bibliothèque `CRlibm`. Cette bibliothèque est un projet logiciel important mené par l'équipe Arénaire depuis plusieurs années. Au moment où Christoph Lauter a commencé sa thèse, il fallait un mois de développement pour ajouter une nouvelle fonction à `CRlibm`. Durant sa thèse, il a développé un prototype de logiciel capable d'effectuer de façon quasi-automatique une telle implémentation. Ce prototype s'appuie intégralement sur Sollya et son langage de script. On peut estimer qu'il suffit à présent de quelques heures pour ajouter une nouvelle fonction à `CRlibm` (une fois les pires cas pour l'arrondi calculés). Un tel succès vient du fait que nous avons toujours pu adapter parfaitement Sollya à nos besoins.

Cela étant, cette souplesse se paie par de nombreux désavantages, au premier rang desquels le besoin permanent de réinventer la roue. Si nous avions pris place dans un logiciel existant, un certain nombre de choses (qui ne relèvent pas de notre domaine de compétence à la base) auraient été fournies d'office : présence d'un shell interactif, gestion de la mémoire, langage de script, structures de données de haut niveau. Nous avons dû refaire tout cela en tâtonnant et en faisant des erreurs en cours de route. Dans les domaines qui ne sont pas les nôtres, nous aurions bénéficié de l'effort de prédécesseurs plus aguerris que nous. En outre, en terme de publicité, nous aurions immédiatement bénéficié de l'effet de marque d'un logiciel déjà installé sur le marché. Tout le monde utilise Matlab ou Maple ; si nous avions conçu Sollya comme une extension d'un de ces logiciels, nous aurions facilité la vie de nos futurs utilisateurs en leur épargnant l'effort d'apprendre à se servir d'un nouveau logiciel.

En outre, nous avons fait, dès le départ, des choix qui constituent à présent un frein au développement du logiciel. En voici quelques exemples.

Dans Sollya les fonctions sont univariées : au départ, nous n'avions aucune raison de vouloir traiter le cas de fonctions multivariées puisque ce n'était pas notre sujet d'étude et que nous voulions simplement bricoler rapidement un prototype d'algorithme. Nous avons donc décidé que les expressions représenteraient exclusivement des fonctions univariées. Ajouter le support des fonctions multivariées à présent est une tâche lourde qui demande de repasser sur la quasi totalité du code.

Sollya est codé en C. C'était un langage adapté pour écrire rapidement un parseur minimal, utiliser les bibliothèques `MPFR` et `MPFI`. Mais le projet grossissant, la pauvreté du langage C (pas de structures de données de haut niveau, mécanisme de gestion des exceptions difficile à utiliser, etc.) est devenue un handicap. À la réflexion, un langage comme C++ laisserait plus de place à la modularité et à la souplesse du codage. Nous aurions à notre disposition une grande variété de bibliothèques performantes.

La structuration du code source est aussi un problème. Dès le début, nous avons pensé Sollya comme un logiciel muni d'un shell interactif. Nous avons donc écrit un outil minimal composé d'un shell à partir duquel on pouvait manipuler quelques algorithmes numériques. Il en résulte une architecture assez monolithique où fonctions numériques et gestion de l'interface sont mélangées. Il aurait fallu séparer clairement les deux aspects dès le début. Au bout d'un moment, nous avons pris conscience que les algorithmes numériques que nous développions pouvaient présenter un intérêt en dehors de Sollya. Il était raisonnable de les rendre accessibles à travers une bibliothèque logicielle autonome permettant à d'autres d'utiliser ces algorithmes dans leur propre programme C. Nous avons donc publié une bibliothèque appelée `libsollya` qui exporte les fonctionnalités utiles de Sollya. Compte-tenu de l'architecture générale du logiciel, nous avons été obligés d'exporter toutes nos structures de données internes. Ce n'est pas souhaitable : si nous voulons les changer dans une version future, nous devrons casser la compatibilité de `libsollya` avec les versions précédentes. Il aurait mieux valu adopter dès le début la démarche inverse : concevoir une bonne bibliothèque numérique et écrire ensuite un shell interactif en utilisant cette bibliothèque. Ceci assure que la bibliothèque est bien pensée et, en outre, cela permet de modifier le shell et la bibliothèque indépendamment l'un de l'autre.

Voici enfin un dernier exemple. Quand nous avons commencé à développer Sollya, nous ne pensions pas du tout que nous ajouterions un jour un langage de script. Nous avions juste besoin d'une sorte de calculette perfectionnée. Mais au bout d'un certain temps, c'est apparu comme un besoin naturel. Or, implémenter un langage de script est une entreprise délicate. Il faut donner à l'utilisateur du langage des structures de données (essentiellement les listes, dans le cas de Sollya ; mais on peut imaginer que d'autres besoins vont se faire sentir), il faut gérer — au moins de façon minimale — un système de types (avec les problèmes de polymorphismes associés), il faut gérer les variables et leur portée, etc. Nous n'avions pas de compétences sérieuses dans ce domaine. Le résultat est donc un langage de script assez peu ergonomique et dont l'interpréteur est inefficace.

## 4.3 Axes de développement futur

Il y a beaucoup de pistes possibles pour le développement futur de Sollya. La première question, et la plus critique, est d'entamer une réflexion sur l'agrégation de nouveaux développeurs. Il n'est pas certain que Christoph Lauter et moi pourrons consacrer autant de temps à Sollya dans les années à venir. Il est donc crucial de recruter de nouveaux développeurs. Depuis peu, Sollya a trouvé un troisième développeur régulier en la personne de Mioara Joldeş qui est, elle aussi, doctorante dans l'équipe Arénaire.

Le défi principal est de réfléchir à un nouveau mode de développement. Jusqu'à présent, nous avons développé le logiciel à deux, et toutes les décisions se prenaient de vive voix, de façon concer-

tée. Nous avions facilement un œil sur le travail de l'autre ce qui permettait de garder une idée précise de l'ensemble du logiciel. Ce fonctionnement ne sera pas durable et il faut donc mettre en place un mode d'organisation différent.

Nombre de petites améliorations ou d'ajouts d'algorithmes sont à l'ordre du jour : programmer un algorithme d'intégration numérique performant et un algorithme d'interpolation polynomiale, par exemple. Par ailleurs, il faut encore ajouter un certain nombre de petites fonctionnalités, élémentaires mais toujours absentes.

Sur le plan scientifique, trois grands chantiers vont demander un réel travail bibliographique et de recherche : nous souhaitons, à moyen terme, ajouter un support minimal de l'algèbre linéaire (résolution de systèmes linéaires, produit de matrices, inversion, décomposition LU, etc.). Dans la ligne philosophique de Sollya, nous pouvons vouloir deux choses :

— ou bien des algorithmes exacts, travaillant sur des matrices entières, mais qui utilisent de l'arithmétique flottante pour être plus performants ;
— ou bien des algorithme approchés, travaillant sur des matrices de nombres en virgule flottante, mais dont le comportement est bien spécifié (en terme d'erreur entre le résultat exact du calcul et le résultat renvoyé).

Il nous faudra probablement faire un choix entre l'une ou l'autre de ces deux conceptions.

Le deuxième chantier est celui de la refonte rigoureuse de l'évaluateur de fonctions présent au cœur de Sollya (qui correspond à la commande `evaluate`). En effet, il s'agit d'une primitive essentielle pour le calcul numérique certifié en précision arbitraire. Cette brique de base prend en entrée une expression $f$, un nombre flottant $x$ de précision arbitraire et une précision cible $t$ et renvoie en sortie un intervalle contenant la valeur exacte $f(x)$ et ne comprenant au plus qu'un seul nombre flottant en précision $t$ (sauf si $f(x) = 0$, auquel cas cette condition ne peut être vérifiée en général). L'algorithme doit adapter automatiquement sa précision au fur et à mesure, de façon à obtenir un intervalle suffisamment fin en sortie. Notre implémentation actuelle est assez naïve alors que des travaux existent sur le sujet. Nous avons l'intention de récrire notre évaluateur en tenant compte de ce savoir-faire.

Le dernier chantier consiste à rajouter le support des fonctions multivariées dans Sollya. Cela pose quelques petites questions ergonomiques mais surtout, comme expliqué plus haut, cela demande de repasser sur la quasi totalité du code source de Sollya.

Enfin, nous pourrions songer, à la lumière des réflexions précédentes, à récrire Sollya de fond en comble en le concevant comme une bibliothèque numérique (qu'il faudra donc spécifier et bien penser) sur laquelle on s'appuie pour créer un shell interactif. Pour ne pas s'encombrer de la gestion d'un langage de script *ad hoc*, nous pourrions songer à intégrer Sollya à un environnement de script déjà existant, comme Python par exemple.

# Conclusion

Dans la présente thèse, nous avons étudié plusieurs questions liées à l'évaluation des fonctions numériques. Plus précisément, nous avons distingué deux problématiques, suivant que l'évaluation se fait en précision arbitraire ou en précision fixée.

Dans un premier temps, nous nous sommes concentrés sur la précision arbitraire. Pour évaluer une fonction $f$ en précision arbitraire en un point $x$, il faut d'abord trouver un procédé numérique convergeant vers $f(x)$. Ce procédé doit converger rapidement bien sûr, mais il faut aussi qu'il soit numériquement stable. Une fois le procédé numérique choisi, il faut réaliser une analyse d'erreur : on borne ainsi les erreurs d'arrondi qui se produisent au cours de l'évaluation. Cette analyse d'erreur doit être faite soigneusement puisque c'est elle qui garantit la qualité du résultat final. On peut alors implémenter effectivement la fonction.

Nous avons étudié au premier chapitre le cas particulier des fonctions erf et erfc. Nous avons proposé trois algorithmes différents :

— le premier utilise la série de Taylor de $\operatorname{erf}(x)$ ;
— le deuxième utilise la série de Taylor de $e^{x^2}\operatorname{erf}(x)$ ;
— le troisième utilise le développement asymptotique de $\operatorname{erfc}(x)$.

Le premier algorithme converge assez rapidement mais est numériquement très mal conditionné pour de grandes valeurs de $x$. Le deuxième corrige ce problème : il converge aussi vite et est numériquement stable. Cependant, il nécessite l'évaluation supplémentaire de $e^{-x^2}$ ce qui le rend moins intéressant pour les petites valeurs de $x$. Enfin, le troisième algorithme est le plus efficace, mais il ne permet pas de fournir des valeurs arbitrairement précises de $\operatorname{erf}(x)$ et $\operatorname{erfc}(x)$. Lorsque la précision est trop grande, il faut donc utiliser l'un des deux autres algorithmes.

Le schéma que nous avons employé pour traiter l'exemple des fonction erf et erfc est assez général et peut servir de cadre pour l'implémentation d'autres fonctions en précision arbitraire.

Dans un deuxième temps, nous avons abordé le problème de l'évaluation de fonctions en précision fixée : on connaît, au moment de l'implémentation, la précision avec laquelle on veut évaluer $f$. On peut bien sûr se contenter d'évaluer $f$ en utilisant un code en précision arbitraire. Mais comme on sait à l'avance quelle précision l'on désire, il est possible de concevoir un programme bien plus efficace.

Une telle implémentation en précision fixée passe, la plupart du temps, par la recherche d'une bonne approximation $p$ de la fonction $f$. En règle générale, moins elle est précise, plus elle est facile à évaluer : on choisit donc l'approximation $p$ pour qu'elle fournisse juste la précision requise. Pour trouver $p$, et pour mesurer l'erreur entre $p$ et $f$, il est nécessaire de calculer de nombreuses valeurs de $f$ en grande précision : ceci justifie le soin apporté à l'implémentation de fonctions en précision arbitraire.

La recherche d'approximation $p$ de bonne qualité a fait l'objet du chapitre 2. Le plus souvent, on cherche $p$ sous la forme d'un polynôme : on peut ainsi l'évaluer en n'utilisant que des additions et des multiplications, et en exploitant le parallélisme des processeurs modernes. La question naturelle est alors la suivante : étant donné un degré $n$, quel polynôme de degré inférieur ou égal à $n$ approche

le mieux la fonction $f$ ? Cette question a été étudiée et résolue entre la fin du XIX$^{\text{e}}$ siècle et la première moitié du XX$^{\text{e}}$. Le polynôme de meilleure approximation est unique, bien caractérisé, et l'algorithme de Rémez permet de le calculer rapidement. La situation se complique lorsqu'on souhaite ajouter d'autres contraintes, comme fixer à l'avance la valeur d'un coefficient par exemple.

Le problème peut être formulé en toute généralité de la façon suivante : étant donnés $n+1$ fonctions continues $(\varphi_0, \ldots, \varphi_n)$ et une fonction continue $g$ à approcher sur un intervalle $[a, b]$, trouver une combinaison linéaire $p$ des $\varphi_j$ qui minimise $\|p-g\|_\infty$. Cette formulation recouvre le cas de l'approximation polynomiale classique, mais aussi de l'approximation « à trous » (c'est-à-dire approximation par des polynômes dont on a fixé la valeur de certains coefficients). En outre, cela permet d'imaginer faire de l'approximation dans des bases inhabituelles (polynômes trigonométriques, combinaisons linéaires de fonctions de la forme $e^{\lambda_j x}$, etc.)

Si les $\varphi_j$ vérifient une condition appelée *condition de Haar*, toute la théorie de l'approximation polynomiale se transpose. Lorsque cette condition n'est pas vérifiée, en revanche, le problème est plus délicat. Nous avons rappelé la théorie, avec et sans condition de Haar, et l'avons illustrée sur des exemples.

Pour pouvoir être effectivement implémenté dans un programme, le polynôme $p$ doit avoir des coefficients représentables sous forme de nombres en virgule fixe ou en virgule flottante. Une approche naïve consiste à penser qu'on peut calculer le polynôme $p^\star$ de meilleure approximation à coefficients réels et arrondir chacun de ses coefficients au nombre machine le plus proche. On obtient ainsi un polynôme $\widehat{p}$ dont les coefficients sont représentables. Cependant, ce procédé peut nettement dégrader la qualité d'approximation de $\widehat{p}$ par rapport à $p^\star$. Nous avons décrit deux algorithmes, complémentaires l'un de l'autre, pour trouver un très bon (voire la liste des meilleurs) polynôme(s) d'approximation dont les coefficients sont des nombres machine :

— le premier repose sur des outils de programmation linéaire. Il permet d'obtenir des informations rigoureuses sur le polynôme $p$ recherché comme la valeur des exposants des coefficients par exemple. Parfois, il est aussi possible de montrer que certains coefficients n'ont qu'une seule valeur possible. En utilisant une stratégie dichotomique, il est possible d'utiliser cet algorithme pour trouver des polynômes quasi-optimaux (dans certains cas, on peut même calculer la liste des polynômes optimaux) ;

— le deuxième algorithme repose sur la théorie des réseaux et l'algorithme LLL. Il permet d'obtenir rapidement des polynômes de très bonne qualité dont les coefficients sont des nombres machine. Il est uniquement heuristique mais ses résultats pratiques sont excellents et peuvent notamment être utilisés pour accélérer l'approche dichotomique.

À la fin du chapitre 2, nous avons rapidement étudié le cas où la fonction d'approximation $p$ est une fraction rationnelle. Si la fraction vérifie certaines conditions, il est possible de l'évaluer efficacement en matériel, et sans faire explicitement de division, à l'aide d'un algorithme appelé *E-méthode*. Nous avons proposé un algorithme (encore imparfait) pour calculer de très bonnes fractions dont les coefficients sont des nombres machine et respectant les contraintes de la E-méthode.

Une fois qu'on a trouvé une approximation $p$ qui soit de bonne qualité et dont les coefficients sont représentables, il ne reste plus qu'à l'implémenter. Tout comme dans le cas de la précision arbitraire, il faut faire une analyse d'erreur, de façon à contrôler la propagation des erreurs d'arrondi durant l'évaluation de $p$. L'outil Gappa développé par Melquiond a permis d'automatiser cette analyse d'erreur et de la prouver dans un langage formel. Cependant, pour démontrer la validité de l'implémentation complète, il ne faut pas seulement borner rigoureusement les erreurs d'arrondi, mais aussi borner l'erreur d'approximation. Il faut trouver un moyen de certifier que l'erreur (absolue ou relative, suivant les applications) entre $p$ et $f$ ne dépasse pas un certain seuil. Cela revient à calculer un encadrement certifié de la norme sup de la fonction d'erreur $\varepsilon = p - f$ ou $\varepsilon = p/f - 1$.

Cette question a été étudiée dans le chapitre 3. Les algorithmes purement numériques qui calculent le maximum d'une fonction ne peuvent pas nous convenir : en effet, ils donnent uniquement

une estimation du maximum. Leur résultat n'est pas certifié. Il existe des algorithmes (l'algorithme de Hansen et ses variantes) capables de donner un encadrement certifié du maximum d'une fonction sur un intervalle. Malheureusement, ces algorithmes se comportent très mal lorsqu'on essaie de les utiliser sur une fonction d'erreur. En effet, ils reposent sur l'utilisation de l'arithmétique d'intervalle : celle-ci ne fonctionne pas bien sur des expressions de la forme $p - f$ ou $p/f - 1$ faisant intervenir la soustraction de deux termes fortement corrélés.

Pour que l'arithmétique d'intervalle donne des résultats pertinents dans ce contexte, il faut l'utiliser sur des intervalles de taille minuscule. La méthode que nous avons proposée consiste à localiser très finement les zéros de la dérivée de $\varepsilon'$ ; ensuite, l'arithmétique d'intervalle peut être utilisée avec succès pour évaluer $\varepsilon$ sur chacun des zéros de $\varepsilon'$ de façon à trouver le maximum de $\varepsilon$ sur $[a, b]$. La difficulté réside dans le fait de garantir qu'on n'a oublié aucun zéro de $\varepsilon'$. Nous nous ramenons au problème de la localisation des zéros d'un polynôme ; on peut alors dénombrer rigoureusement les racines comprises dans l'intervalle $[a, b]$.

## De `CRlibm` à Sollya

Une grande partie des travaux présentés dans cette thèse tire son origine de problématiques liées au développement de la bibliothèque `CRlibm`. L'implémentation d'une fonction $f$ passe souvent par une importante phase d'expérimentation. On ne sait généralement pas à l'avance quelle structure donner au polynôme d'approximation : va-t-on ou non fixer la valeur d'un coefficient, quelle précision va-t-on utiliser pour représenter chaque coefficient ? Lorsqu'il y a une réduction d'argument pour $f$, on ne sait même pas, en général, sur quel intervalle approcher la fonction :

— va-t-on choisir une réduction d'argument coûteuse et une approximation polynomiale sur un tout petit intervalle ?

— ou bien une réduction d'argument rapide mais une approximation sur un grand intervalle ?

Pour tester chaque hypothèse, il faut calculer des polynômes de meilleure approximation, dessiner des graphes d'erreur, calculer des normes sup, etc.

C'est pour faciliter ces expérimentations que nous avons écrit le logiciel Sollya. Au début, il s'agissait d'un outil sommaire avec peu de fonctionnalités. Au fur et à mesure de nos besoins, nous l'avons enrichi, lui ajoutant en particulier un langage de script. La philosophie du logiciel s'est forgée peu à peu : avec Sollya, on indique la précision avec laquelle on souhaite connaître le résultat d'un calcul. La précision intermédiaire nécessaire est déterminée automatiquement. L'utilisateur n'a pas à se soucier des difficultés numériques : la valeur affichée par le logiciel n'est pas une simple estimation numérique, mais une valeur dont la précision est garantie.

La portée de Sollya a désormais dépassé le simple cadre du développement de `CRlibm` : il a vocation à fournir un environnement sûr et efficace à toute personne désirant faire des calculs numériques. Il est déjà utilisé en dehors du cercle réduit de l'équipe Arénaire où il a été conçu. Les axes de développement futur sont nombreux et porteurs de questions de recherche intéressantes :

— ajout de procédures pour faire de l'interpolation polynomiale, de l'intégration numérique ;

— résolution numérique d'équations différentielles ;

— support minimal de l'algèbre linéaire (résolution de systèmes linéaires, inversion de matrices, recherche des valeurs propres d'une application linéaire) ;

— support de fonctions multivariées (norme sup de fonctions multivariées, polynôme de meilleure approximation à plusieurs variables) ;

— support de fonctions à variable(s) complexe(s).

Bien sûr, ces problèmes devront être étudiés dans un double souci d'efficacité et d'adaptation automatique de la précision pour garantir la qualité du résultat. Des algorithmes existent déjà pour résoudre certaines de ces questions, mais il n'existe pas de logiciel les regroupant dans un cadre unifié et ergonomique. Sollya pourrait devenir un tel logiciel.

## Automatisation de la synthèse de code

L'implantation d'une fonction numérique (que ce soit en précision fixée ou en précision arbitraire) est une tâche longue et parfois fastidieuse. L'analyse d'erreur est délicate : les risques de se tromper dans l'estimation d'une erreur sont grands. La moindre faute peut compromettre la qualité finale de l'implémentation. Tout le monde garde en mémoire le bug du Pentium : une erreur stupide, commise au cours du développement de l'algorithme de division, a coûté très cher (en dollars et en réputation) à la société Intel.

Il existe souvent de nombreuses stratégies différentes pour évaluer une même fonction $f$ : rien que pour les fonctions erf et erfc, nous avons vu trois méthodes différentes, et d'autres algorithmes seraient envisageables (utilisation du développement en fraction continue ou évaluation par scindage binaire, pour n'en citer que deux). Dans le cadre de l'évaluation en précision fixée, il faut déterminer le compromis entre degré de l'approximation polynomiale et réduction d'argument.

Compte tenu du travail que cela représente, il est difficilement envisageable d'implémenter des dizaines d'algorithmes pour voir lequel est le plus efficace en pratique. Il apparaît donc de plus en plus indispensable d'automatiser (tout ou partie) du processus de développement d'une fonction. Une telle automatisation a un double avantage :

— le développeur, débarrassé de la partie longue et difficile, peut explorer de nouvelles voies dans lesquelles il ne se serait pas aventuré : lorsqu'il faut un mois pour concevoir et implémenter une fonction, on ne se risque pas à essayer un algorithme dont l'efficacité n'est pas certaine. Lorsque l'implémentation est produite automatiquement en quelques minutes, il devient possible de tester ces pistes de recherche incertaines ;

— le code généré est considérablement plus sûr : on peut souvent envisager de prouver automatiquement la correction du code en même temps qu'on le génère. Cette preuve peut être vérifiée par un outil formel. Une erreur dans le code final devient alors très improbable.

Durant sa thèse, Lauter a écrit un ensemble de scripts [1] (utilisant Sollya) qui permettent d'automatiser une grande partie de l'implantation de fonctions en précision fixée. Concrètement, l'utilisateur donne une fonction, un intervalle et une précision cible et l'outil de Lauter produit automatiquement (en quelques minutes) un fichier écrit en langage C qui implémente la fonction. En outre, l'outil de Lauter utilise Gappa pour fournir une analyse d'erreur qui peut être prouvée formellement avec l'assistant de preuve COQ. Pour parvenir à ce résultat, le logiciel de Lauter s'appuie (entre autres) sur tous les algorithmes et outils exposés dans cette thèse.

Les résultats présentés aux chapitres 2 et 3 ont largement dépassé le cadre initial des problématiques de `CRlibm`. Ils peuvent être utiles à n'importe qui souhaitant faire du calcul approché de fonctions. Les libms ont un gros défaut : elles ne proposent qu'un certain nombre de fonctions de base, choisies par avance. En outre, toutes les fonctions de base sont approchées avec la précision de 53 bits correspondant au format `double`. Si, dans un programme, quelqu'un a besoin d'évaluer $\sin(x)/\ln(1+x)$ avec 40 bits de précision, le code C a la forme suivante :

```
a = sin(x);
b = log(1+x);
return a/b;
```

Ce programme fait donc deux évaluations de fonctions en précision `double` : chacune prend du temps et le résultat final est beaucoup trop précis.

Avec les algorithmes présentés dans cette thèse, il est envisageable de produire directement du code pour évaluer $\sin(x)/\ln(1+x)$ avec une précision de 40 bits : il n'est pas improbable que le temps d'exécution soit divisé par deux. Les outils proposés sont d'ailleurs suffisamment généraux

1. Il s'agit de μετά-libm : distribué sous licence LGPL et disponible à l'adresse http://lipforge.ens-lyon.fr/www/metalibm/. Il ne s'agit que d'un prototype en phase encore très expérimentale.

pour permettre de sortir des sentiers battus de l'approximation polynomiale classique : on peut imaginer utiliser des bases exotiques comme $(1, x, x^2, e^x)$, ce qui est certainement intéressant pour approcher des fonctions dont la croissance est très rapide par exemple (surtout si on a déjà eu à évaluer $e^x$ avant, pour une raison quelconque).

Ce travail d'automatisation, effectué pour l'approximation en précision fixée, n'a pas encore été fait pour la précision arbitraire. Les techniques d'approximation en précision arbitraire étant assez différentes de celles utilisées en précision fixée, les algorithmes d'automatisation développés jusqu'ici ne sont pas pertinents pour ce nouveau problème. Cela étant, la méthodologie et le savoir faire acquis durant cette thèse devraient pouvoir profiter aussi à la précision arbitraire.

L'automatisation du développement de code en précision arbitraire devrait présenter les mêmes avantages que pour la précision fixée : gain de temps dans l'implémentation, sûreté du code généré (avec possibilité de générer une preuve formelle), possibilité de comparer des algorithmes qu'on n'aurait jamais implémentés sans l'automatisation. Pour y parvenir, il faudra répondre à des questions très intéressantes, à la frontière avec le calcul formel : estimation automatique de la vitesse de convergence d'une suite, minoration et majoration fines de restes de séries, estimations fines et rigoureuses de la valeur de certaines fonctions auxiliaires, etc. Des travaux récents [MS09] apportent déjà quelques éléments de réponse à ces questions.

## Vers plus de sûreté

Doit-on, comme Householder, redouter de monter dans les avions parce que ceux-ci sont conçus en utilisant l'arithmétique flottante ? Il est si facile de prendre pour vrai un résultat numérique sur lequel on n'a, en réalité, aucune garantie. L'arithmétique flottante tend tellement de pièges aux imprudents qui lui font confiance…

Mais le modèle de la virgule flottante n'est pas en cause. Il n'est tout simplement pas possible de faire des calculs exacts en temps fini avec les nombres réels ; quel que soit le modèle utilisé pour représenter les nombres, ce sera toujours un modèle approché, et la plupart des opérations donneront lieu à des erreurs d'arrondi. Ce sont ces erreurs d'arrondi qui sont la cause des comportements déroutant, parfois complètement faux de l'arithmétique flottante.

Les erreurs ne sont pas un problème en soi : nous n'avons jamais besoin de connaître exactement un résultat. Souvent les dix premiers chiffres significatifs sont amplement suffisants. Dans tout le manuscrit, nous avons manipulé des nombres approchés, des polynômes d'approximations, des bornes d'erreurs ; nous avons pu raisonner rigoureusement avec ces objets. Pour le calcul certifié de la norme sup d'une fonction d'erreur $\varepsilon$, c'est même en acceptant de remplacer $\varepsilon'$ par un polynôme d'approximation $T$ que nous avons pu obtenir un encadrement fin et certifié.

Ce qui peut causer du tort, en revanche, c'est d'ignorer les erreurs d'arrondi. C'est de les laisser s'accumuler sans jamais s'en soucier, jusqu'à ce que les valeurs numériques manipulées n'aient plus de sens. Doit-on enseigner à tous les ingénieurs, à tous les numériciens, les subtilités du modèle flottant et de l'analyse d'erreur ? Il faut bien sûr leur en donner quelques notions, mais l'analyse d'erreur n'est pas leur métier ; lorsqu'un ingénieur veut résoudre une équation différentielle pour dessiner le profil d'une aile d'avion, ce dont il a besoin, c'est d'un résultat dans lequel il puisse avoir confiance.

Il faut travailler à la conception d'algorithmes sûrs, qui adaptent leur précision automatiquement pour fournir un résultat certifié. Lorsqu'un ingénieur veut résoudre un système linéaire, il devrait toujours spécifier avec quelle précision il souhaite connaître son résultat. Une valeur numérique seule n'a aucun sens : elle devrait systématiquement être accompagnée d'une borne d'erreur. En tout état de cause, il vaut souvent mieux dire « je ne sais pas » que de renvoyer une valeur arbitraire. Sollya est un pas dans cette direction ; la synthèse automatique de code en est un autre. J'espère les prolonger à l'avenir.

# Bibliographie

Les pages indiquées entre crochets à la suite de chaque référence correspondent aux endroits où il en est fait mention dans le présent manuscrit.


[AI85] American National Standards Institute (ANSI) et Institute of Electrical and Electronic Engineers (IEEE), *IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std754-1985)*, 1985. [page 35]

[AI08] American National Standards Institute (ANSI) et Institute of Electrical and Electronic Engineers (IEEE), *IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std754-2008)*, 2008. Révision de la norme IEEE Std754-1985. [page 35]

[Ajt98] M. Ajtai, *The Shortest Vector Problem in $L_2$ is NP-hard for Randomized Reductions (Extended Abstract).* Dans *STOC*, pages 10–19, 1998. [page 117]

[AS65] M. Abramowitz et I. A. Stegun, *Handbook of Mathematical Functions.* Dover, 1965. [page 40]

[Bab86] L. Babai, *On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica*, 6(1) : 1–13, 1986. [page 124]

[Bak75] A. Baker, *Transcendental Number Theory.* Cambridge Mathematical Library, 1975. [pages 38 et 78]

[BC07] N. Brisebarre et S. Chevillard, *Efficient polynomial $L^\infty$-approximations.* Dans P. Kornerup et J. M. Muller (éditeurs) : *18th IEEE SYMPOSIUM on Computer Arithmetic*, pages 169–176, Los Alamitos, CA, 2007. IEEE Computer Society. [page 30]

[BCE+08] N. Brisebarre, S. Chevillard, M. D. Ercegovac, J. M. Muller et S. Torres, *An Efficient Method for Evaluating Polynomial and Rational Function Approximations.* Dans *ASAP 08, Conference Proceedings, IEEE 19th International Conference on Application-Specific Systems, Architectures and Processors*, pages 245–250, Los Alamitos, CA, 2008. IEEE Computer Society. [page 30]

[BE95] P. Borwein et T. Erdelyi, *Polynomials And Polynomial Inequalities.* Graduate Texts in Mathematics. Springer, 1995. [pages 85 et 87]

[BE96] P. Borwein et T. Erdelyi, *The Integer Chebyshev Problem. Mathematics of Computation*, 65(214) : 661–681, 1996. [page 110]

[BHLT] D. H. Bailey, Y. Hida, X. S. Li et B. Thompson, *ARPREC : An Arbitrary Precision Computation Package.* Le logiciel et sa documentation sont disponibles à l'adresse http://crd.lbl.gov/~dhbailey/mpdist/. [page 38]

[BK75] R. P. Brent et H. T. Kung, *$\mathcal{O}\left((n \log n)^{3/2}\right)$ Algorithms for Composition and Reversion of Power Series.* Dans J. F. Traub (éditeur) : *Analytic Computational Complexity*, pages 217–225, New York, 1975. Academic Press. [page 157]

[BK78] R. P. BRENT et H. T. KUNG, *Fast Algorithms for Manipulating Formal Power Series. Journal of the ACM*, 25(4) : 581–595, 1978. [page 157]

[BMT06] N. BRISEBARRE, J. M. MULLER et A. TISSERAND, *Computing Machine-efficient Polynomial Approximations. ACM Transactions on Mathematical Software*, 32(2) : 236–256, 2006. [pages 30 et 110]

[Bor05] E. BOREL, *Leçons sur les fonctions de variables réelles*, chapitre 4 : représentation des fonctions continues. Gauthier-Villars, 1905. [page 82]

[Bre76] R. P. BRENT, *The Complexity of Multiple-Precision Arithmetic. The Complexity of Computational Problem Solving*, pages 126–165, 1976. [pages 34, 62 et 66]

[Bre78] R. P. BRENT, *A Fortran Multiple-Precision Arithmetic Package. ACM Transactions on Mathematical Software (TOMS)*, 4(1) : 57–70, 1978. [pages 34, 38 et 41]

[Bre80] R. P. BRENT, *Unrestricted Algorithms for Elementary and Special Functions*. Dans S. LAVINGTON (éditeur) : *Information Processing 80 : Proceedings of IFIP Congress 80*, pages 613–619. North-Holland, 1980. [pages 34, 38 et 77]

[Bre02] R. P. BRENT, *Algorithms for Minimization Without Derivatives*. Dover Publications, deuxième édition, 2002. [page 145]

[Bro96] Fabrizio BROGLIA (éditeur), *Lectures in Real Geometry*, tome 23 de *Expositions in Mathematics*, pages 1–67. de Gruyter, 1996. [page 163]

[BS97] C. BENDSTEN et O. STAUNING, *TADIFF, a Flexible C++ Package for Automatic Differentiation Using Taylor Series*. Rapport de recherche 1997-x5-94, Technical University of Denmark, 1997. [pages 157 et 159]

[BT04] J. P. BERRUT et L. N. TREFETHEN, *Barycentric Lagrange Interpolation. SIAM Review*, 46(3) : 501–517, 2004. [page 98]

[Cai99] J. Y. CAI, *Some Recent Progress on the Complexity of Lattice Problems*. Dans *Fourteenth Annual IEEE Conference on Computational Complexity. Proceedings.*, pages 158–178, 1999. [page 117]

[Cas97] J. W. S. CASSELS, *An Introduction to the Geometry of Numbers*. Classics in Mathematics. Springer, 1997. [pages 115, 116 et 119]

[CC90] D. V. CHUDNOVSKY et G. V. CHUDNOVSKY, *Computer Algebra in the Service of Mathematical Physics and Number Theory*. Dans D. V. CHUDNOVSKY et R. D. JENKS (éditeurs) : *Computers in Mathematics*, tome 125 de *Lecture notes in pure and applied mathematics*, pages 109–232. Dekker, 1990. [page 66]

[CGH+96] R. M. CORLESS, G. H. GONNET, D. E. G. HARE, D. J. JEFFREY et D. E. KNUTH, *On the Lambert W function. Advances in Computational Mathematics*, 5(4) : 329–359, 1996. [page 45]

[Che82] E. W. CHENEY, *Introduction to Approximation Theory*. AMS Chelsea Publishing, 1982. [pages 80, 82, 94, 98, 100, 127, 135, 136 et 162]

[Che06] S. CHEVILLARD, *Polynômes de meilleure approximation à coefficients flottants*. Mémoire de master, École normale supérieure de Lyon, 46 allée d'Italie, 69 364 Lyon Cedex 07, 2006. Disponible à l'adresse `http://www.ens-lyon.fr/LIP/Pub/Rapports/DEA/DEA2006/DEA2006-03.ps.gz`. [page 30]

[Che09] S. CHEVILLARD, *The functions* erf *and* erfc *computed with arbitrary precision*. Rapport de recherche RR2009-04, laboratoire de l'informatique du parallélisme (LIP), 46, allée d'Italie, 69 364 Lyon Cedex 07, 2009. Disponible à l'adresse `https://hal.archives-ouvertes.fr/ensl-00356709`. [page 29]

[CJJL] S. CHEVILLARD, M. JOLDEŞ, N. JOURDAN et Ch. LAUTER, *User's Manual of the Sollya Tool*. Disponible à l'adresse http://sollya.gforge.inria.fr/. [pages 31, 67 et 129]

[CJL09] S. CHEVILLARD, M. JOLDEŞ et Ch. LAUTER, *Certified and fast computation of supremum norms of approximation errors*. Dans J. D. BRUGUERA, M. CORNEA, D. DASSARMA et J. HARRISON (éditeurs) : *19th IEEE SYMPOSIUM on Computer Arithmetic*, pages 169–176, Los Alamitos, CA, 2009. IEEE Computer Society. [page 31]

[CL76] C. CARASSO et P. J. LAURENT, *Un algorithme général pour l'approximation au sens de Tchebycheff de fonctions bornées sur un ensemble quelconque*. Dans A. DOLD et B. ECKMANN (éditeurs) : *Approximation Theory*, tome 556 de *Lecture Notes in Mathematics*, pages 99–121. Springer, 1976. [page 94]

[CL07] S. CHEVILLARD et Ch. LAUTER, *A certified infinite norm for the implementation of elementary functions*. Dans A. MATHUR, W. E. WONG et M. F. LAU (éditeurs) : *QSIC 2007, Proceedings of the Seventh International Conference on Quality Software*, pages 153–160, Los Alamitos, CA, 2007. IEEE Computer Society. Une version longue est disponible sous forme du rapport de recherche RR2007-26, laboratoire de l'informatique du parallélisme, à l'adresse https://hal.archives-ouvertes.fr/ensl-00119810/. [pages 31, 150, 151, 152 et 166]

[Coh93] H. COHEN, *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics. Springer, 1993. [pages 118 et 123]

[CR08] S. CHEVILLARD et N. REVOL, *Computation of the error function erf in arbitrary precision with correct rounding*. Dans J. D. BRUGUERA et M. DAUMAS (éditeurs) : *RNC 8 Proceedings, 8th Conference on Real Numbers and Computers*, pages 27–36, 2008. [page 29]

[Dan51] G. B. DANTZIG, *Maximization of a linear function of variables subject to linear inequalities*. Dans Tj. C. KOOPMANS (éditeur) : *Activity Analysis of Production and Allocation*, pages 339–347. John Wiley & Sons Ltd., 1951. [page 105]

[DdD02] D. DEFOUR et F. de DINECHIN, *Software carry-save for fast multiple-precision algorithms*. Dans *35th International Congress of Mathematical Software*, pages 29–40, Beijing, China, 2002. [page 79]

[Def03] D. DEFOUR, *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. Thèse de doctorat, École normale supérieure de Lyon (ÉNS Lyon), 46, allée d'Italie, 69 007 Lyon, 2003. [page 76]

[Dek71] T. J. DEKKER, *A floating-point technique for extending the available precision*. *Numerische Mathematik*, 18(3) : 224–242, 1971. [page 79]

[Des60] J. DESCLOUX, *Contribution au calcul des approximations de Tschebischeff*. Thèse de doctorat, École polytechnique fédérale de Lausanne (EPFL), 1960. [pages 94 et 100]

[Des61] J. DESCLOUX, *Dégénérescence dans les approximations de Tschebyscheff linéaires et discrètes*. *Numerische Mathematik*, 3(1) : 180–187, 1961. [page 94]

[Des86] R. DESCOMBES, *Éléments de théorie des nombres*. PUF, 1986. [page 117]

[DY93] J. P. DEDIEU et J. C. YAKOUBSOHN, *Computing the real roots of a polynomial by the exclusion algorithm*. *Numerical Algorithms*, 4(1) : 1–24, 1993. [page 163]

[Erc75] M. D. ERCEGOVAC, *A general method for evaluation of functions and computation in a digital computer*. PhD thesis, Department of Computer Science, University of Illinois, 1975. [page 130]

[Erc77] M. D. ERCEGOVAC, *A general hardware-oriented method for evaluation of functions and computations in a digital computer*. *IEEE Transactions on Computers*, C-26(7) : 667–880, 1977. [page 130]

[FHL+07] L. FOUSSE, G. HANROT, V. LEFÈVRE, P. PÉLISSIER et P. ZIMMERMANN, *MPFR : A Multiple-Precision Binary Floating-Point Library With Correct Rounding. ACM Transactions on Mathematical Software (TOMS)*, 33(2), 2007. [pages 34, 36 et 62]

[For70] G. E. FORSYTHE, *Pitfalls in computation, or why a math book isn't enough. American Mathematical Monthly*, 77(9) : 931–956, 1970. [page 40]

[GL87] P. M. GRUBER et C. G. LEKKERKERKER, *Geometry of Numbers*. North-Holland Mathematical Library, 1987. [page 115]

[Gri00] A. GRIEWANK, *Evaluating Derivatives - Principles and Techniques of Algorithmic Differentiation*. Frontiers in Applied Mathematics. SIAM, Society for Industrial and Applied Mathematics, 2000. [pages 157 et 159]

[Han79] E. R. HANSEN, *Global Optimization Using Interval Analysis : The One-Dimensional Case. Journal of Optimization Theory and Applications*, 29(3) : 331–344, 1979. [page 151]

[HH08] A. HOORFAR et M. HASSANI, *Inequalities on the Lambert W function and hyperpower function. Journal of Inequalities in Pure and Applied Mathematics*, 9(2) : 1–5, 2008. [page 45]

[Hig02] N. J. HIGHAM, *Accuracy and Stability of Numerical Algorithms*. SIAM, deuxième édition, 2002. [page 43]

[HS97] L. HABSIEGER et B. SALVY, *On Integer Chebyshev Polynomials. Mathematics of Computation*, 66(218) : 763–770, 1997. [page 110]

[HW04] E. R. HANSEN et G. W. WALSTER, *Global Optimization Using Interval Analysis*. Marcel Dekker, deuxième édition, 2004. [page 145]

[Jea00] E. JEANDEL, *Évaluation rapide de fonctions hypergéométriques*. Rapport de recherche 242, Institut national de recherche en informatique et en automatique (INRIA), 2000. Disponible à l'adresse `http://www.inria.fr/rrrt/rt-0242.html`. [pages 34 et 62]

[Kea96] R. B. KEARFOTT, *Rigorous Global Search : Continuous Problems*. Kluwer, 1996. [page 145]

[Knu98] D. E. KNUTH, *The Art of Computer Programming*, tome 2. Addison-Wesley, troisième édition, 1998. [pages 78 et 130]

[Kod80] D. M. KODEK, *Design of optimal finite wordlength FIR digital filters using integer programming techniques. IEEE Transactions on Acoustics Speech and Signal Processing*, ASSP-28(3) : 304–398, 1980. [page 110]

[Kod02] D. M. KODEK, *An approximation error lower bound for integer polynomial minimax approximation. Elektrotehniški. vestnik*, 69(5) : 266–272, 2002. [page 110]

[Lau05] Ch. LAUTER, *Basic building blocks for a triple-double intermediate format*. Rapport de recherche 5702, Institut national de recherche en informatique et en automatique (INRIA), 2005. [page 79]

[Lau08] Ch. LAUTER, *Arrondi correct de fonctions mathématiques – Fonctions univariées et bivariées, certification et automatisation*. Thèse de doctorat, École normale supérieure de Lyon (ÉNS Lyon), 46, allée d'Italie, 69 007 Lyon, 2008. [pages 76 et 78]

[Leb65] N. N. LEBEDEV, *Special Functions & Their Applications*. Prentice-Hall, 1965. [page 37]

[Lef00] V. LEFÈVRE, *Moyens arithmétiques pour un calcul fiable*. Thèse de doctorat, École normale supérieure de Lyon, 46, allée d'Italie, 69 007 Lyon, 2000. [page 78]

[LLL82] A. K. LENSTRA, H. W. LENSTRA JR et L. LOVÁSZ, *Factoring Polynomials with Rational Coefficients. Mathematische Annalen*, 261(4) : 515–534, 1982. [page 118]

[Lov86] L. LOVÁSZ, *An Algorithmic Theory of Numbers, Graphs, and Convexity*. CBMS-NSF Regional Conference Series in Applied Mathematics, volume 50. SIAM, 1986. [pages 118, 120 et 121]

[LSZ03] V. LEFÈVRE, D. STEHLÉ et P. ZIMMERMANN, *Worst Cases and Lattice Reduction*. Dans J. C. BAJARD et M. SCHULTE (éditeurs) : *16th IEEE SYMPOSIUM on Computer Arithmetic*, pages 142–147. IEEE, 2003. [page 115]

[LSZ08] V. LEFÈVRE, D. STEHLÉ et P. ZIMMERMANN, *Worst Cases for the Exponential Function in the IEEE 754r decimal64 Format*. Dans *Reliable Implementation of Real Number Algorithms : Theory and Practice*, tome 5045 de *Lecture Notes in Computer Science*, pages 114–126. Springer-Verlag, 2008. [page 78]

[MB03] K. MAKINO et M. BERZ, *Taylor Models and other Validated Functional Inclusion Methods*. *International Journal of Pure and Applied Mathematics*, 4(4) : 379–456, 2003. [page 162]

[Mel06] G. MELQUIOND, *De l'arithmétique d'intervalles à la certification de programmes*. Thèse de doctorat, École normale supérieure de Lyon (ÉNS Lyon), 46, allée d'Italie, 69 007 Lyon, 2006. [pages 76 et 144]

[Mel08] G. MELQUIOND, *Floating-point arithmetic in the Coq system*. Dans J. D. BRUGUERA et M. DAUMAS (éditeurs) : *RNC 8 Proceedings, 8th Conference on Real Numbers and Computers*, pages 93–102, 2008. [page 167]

[Mes97] F. MESSINE, *Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution des problèmes avec contraintes*. Thèse de doctorat, Institut national polytechnique de Toulouse, 1997. [page 151]

[Mic01] D. MICCIANCIO, *The Shortest Vector in a Lattice is Hard to Approximate to within Some Constant*. *SIAM Journal on Computing*, 30(6) : 2008–2035, 2001. [page 117]

[Moo79] R. E. MOORE, *Methods and Applications of Interval Analysis*. SIAM Studies in Applied Mathematics, 1979. [pages 70, 157 et 159]

[MS09] M. MEZZAROBBA et B. SALVY, *Effective Bounds for P-Recursive Sequences*. Rapport de recherche abs/0904.2452, arXiv, 2009. [page 177]

[Mul89] J. M. MULLER, *Arithmétique des ordinateurs - opérateurs et fonctions élémentaires*. Études et recherches en informatique. Masson, 1989. Disponible au téléchargement à l'adresse https://hal.archives-ouvertes.fr/ensl-00086707. [pages 34 et 133]

[Mul06] J. M. MULLER, *Elementary Functions, Algorithms and Implementation*. Birkhäuser Boston, deuxième édition, 2006. [page 94]

[Mü00] N. Th. MÜLLER, *The iRRAM : Exact Arithmetic in C++*. Dans J. BLANCK, V. BRATTKA et P. HERTLING (éditeurs) : *4th International Workshop, CCA 2000 Swansea, UK, September 17-19, 2000 Selected Papers*, tome 2064 de *Lecture Notes In Computer Science*, pages 222–252. Springer, 2000. [page 70]

[Neh01] M. NEHER, *Validated Bounds for Taylor Coefficients of Analytic Functions*. *Reliable Computing*, 7(4) : 307–319, 2001. [page 162]

[Neh03] M. NEHER, *Improved Validated Bounds for Taylor Coefficients and for Taylor Remainder Series*. *Journal of Computational and Applied Mathematics*, 152(1–2) : 393–404, 2003. [page 162]

[Neu90] A. NEUMAIER, *Interval Methods for Systems of Equations*. Cambridge University Press, 1990. [pages 70 et 72]

[Neu03] A. NEUMAIER, *Taylor Forms–Use and Limits*. *Reliable Computing*, 9(1) : 43–79, 2003. [page 162]

[NS01] P. Q. NGUYEN et J. STERN, *The Two Faces of Lattices in Cryptology. Lecture Notes in Computer Science*, 2146 : 146–180, 2001. [page 115]

[NS04] P. NGUYEN et D. STEHLÉ, *Low-Dimensional Lattice Basis Reduction Revisited.* Dans *Proceedings of the 6th Algorithmic Number Theory Symposium (ANTS VI)*, tome 3076 de *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, 2004. [page 118]

[Odl91] A. M. ODLYZKO, *The Rise and Fall of Knapsack Cryptosystems.* Dans *PSAM : Proceedings of the 42th Symposium in Applied Mathematics, American Mathematical Society*, 1991. [page 115]

[Pow81] M. J. D. POWELL, *Approximation theory and methods.* Cambridge University Press, 1981. [pages 135 et 136]

[PS73] M. S. PATERSON et L. J. STOCKMEYER, *On the number of nonscalar multiplications necessary to evaluate polynomials. SIAM Journal on Computing*, 2(1) : 60–66, 1973. [page 41]

[PT08] R. PACHÓN et L. N. TREFETHEN, *Barycentric-Remez algorithms for best polynomial approximation in the chebfun system.* Rapport de recherche NA-08/20, Oxford University Computing Laboratory, 2008. [page 98]

[Rem34] E. REMES, *Sur le calcul effectif des polynomes d'approximation de Tchebichef. Comptes rendus hebdomadaires des séances de l'Académie des Sciences*, 199 : 337–340, 1934. [page 94]

[Rev06] G. REVY, *Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants.* Mémoire de master, École normale supérieure de Lyon, 46, allée d'Italie, 69 007 Lyon, 2006. Disponible à l'adresse `https://hal.archives-ouvertes.fr/ensl-00119498/`. [pages 78 et 130]

[RR88] H. RATSCHEK et J. ROKNE, *New Computer Methods for Global Optimization.* John Wiley, 1988. [page 145]

[RZ01] F. ROUILLIER et P. ZIMMERMANN, *Efficient Isolation of a Polynomial Real Roots.* Rapport de recherche 4113, Institut national de recherche en informatique et en automatique (INRIA), 2001. [page 163]

[RZ04] F. ROUILLIER et P. ZIMMERMANN, *Efficient isolation of polynomial's real roots. Journal of Computational and Applied Mathematics*, 162(1) : 33–50, 2004. [page 163]

[Sch86] A. SCHRIJVER, *Theory of Linear and Integer Programming.* Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons Ltd., 1986. [pages 105 et 136]

[Smi89] D. M. SMITH, *Efficient Multiple-Precision Evaluation of Elementary Functions. Mathematics of Computation*, 52(185) : 131–134, 1989. [pages 39 et 41]

[Ste98a] J. STERN, *Lattices and Cryptography : An Overview. Lecture Notes in Computer Science*, 1431 : 50–53, 1998. [page 115]

[Ste98b] G. W. STEWART, *Afternotes goes to Graduate School - Lectures on Advanced Numerical Analysis.* Society for Industrial and Applied Mathematics (SIAM), 1998. [pages 82 et 83]

[Sti59] E. STIEFEL, *Über diskrete und lineare Tschebyscheff-Approximationen. Numerische Mathematik*, 1(1) : 1–28, 1959. [pages 93, 95, 96, 97, 99 et 100]

[Sti60] E. STIEFEL, *Note on Jordan elimination, linear programming and Tchebycheff approximation. Numerische Mathematik*, 2(1) : 1–17, 1960. [page 94]

[Sti64] E. STIEFEL, *Methods — old and new — for solving the Tchebycheff approximation problem. Journal of the Society for Industrial and Applied Mathematics : Series B, Numerical Analysis*, 1 : 164–176, 1964. [page 98]

[Stu35] C. STURM, *Mémoire sur la résolution des équations numériques. Sciences mathématiques et physiques*, 6 : 271–318, 1835. Mémoires présentés par divers savans à l'Académie royale des sciences de l'Institut de France. [page 163]

[Tch59] P. L. TCHEBYCHEFF, *Sur les questions de minima qui se rattachent à la représentation approximative des fonctions. Mémoires de l'Académie impériale des Sciences de Saint-Pétersbourg*, 1859. [page 82]

[vdH07] J. van der HOEVEN, *On Effective Analytic Continuation. Mathematics in Computer Science*, 1(1) : 111–175, 2007. [page 162]

[Vei60] L. VEIDINGER, *On the Numerical Determination of the Best Approximations in the Chebyshev Sense. Numerische Mathematik*, 2(1) : 99–105, 1960. [pages 94 et 98]

[Wat74] G. A. WATSON, *The Calculation of Best Restricted Approximations . SIAM Journal on Numerical Analysis*, 11(4) : 693–699, 1974. [page 94]

[Wu02] Q. WU, *On the Linear Independence Measure of Logarithms of Rational Numbers. Mathematics of Computation*, 72(242) : 901–911, 2002. [page 110]

[Ziv91] A. ZIV, *Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit. ACM Transactions on Mathematical Software (TOMS)*, 17(3) : 410–423, 1991. [page 38]

**Résumé**

Les systèmes informatiques permettent d'évaluer des fonctions numériques telles que $f = \exp$, sin, arccos, etc. Cette thèse s'intéresse au processus d'implémentation de ces fonctions. Suivant la cible visée (logiciel ou matériel, faible ou grande précision), les problèmes qui se posent sont différents, mais l'objectif est toujours d'obtenir l'implémentation la plus efficace possible. Nous étudions d'abord, à travers un exemple, les problèmes qui se posent dans le cas où la précision est arbitraire.

Lorsque, à l'inverse, la précision est connue d'avance, la fonction $f$ est souvent remplacée par un polynôme d'approximation $p$. Un tel polynôme peut ensuite être évalué très efficacement en machine. En pratique, les coefficients de $p$ doivent être représentables sur un nombre fini donné de bits. Nous proposons un ensemble d'algorithmes (certains sont heuristiques, d'autres rigoureux) pour trouver de très bons polynômes d'approximation répondant à cette contrainte. Ces résultats s'étendent au cas où la fonction d'approximation est une fraction rationnelle. Une fois $p$ trouvé, il faut prouver que l'erreur $|p - f|$ n'excède pas un certain seuil. La nature particulière de la fonction $p - f$ (soustraction de deux fonctions très proches) rend cette propriété difficile à prouver rigoureusement. Nous proposons un algorithme capable de contourner cette difficulté.

Tous ces algorithmes ont été intégrés au logiciel Sollya, développé pendant la thèse. À l'origine conçu pour faciliter l'implémentation de fonctions, ce logiciel s'adresse à présent à toute personne souhaitant faire des calculs numériques dans un cadre complètement fiable.

**Mots-clés :** approximation polynomiale, arithmétique en virgule flottante, fonctions numériques, erreurs d'arrondi, norme sup, réseau euclidien, bibliothèque logicielle, précision arbitraire.

---

**Abstract**

With computers, it is possible to evaluate some numerical functions such as $f = \exp$, sin, arccos, etc. The purpose of this thesis is to study how these functions can be implemented. Depending on the target architecture (software or hardware, small or high accuracy required), different problems must be addressed, but the final goal is always to eventually obtain an implementation as efficient as possible. We first study with an example the problems that arise in the case when the precision is arbitrary.

When, on the contrary, the precision is known in advance, the function $f$ is often replaced by an approximation polynomial $p$. Such a polynomial can then be very efficiently evaluated. In practice, the coefficients of $p$ must be representable on a given finite number of bits. We propose several algorithms (some of them are heuristic and others are rigorous) for finding very good approximation polynomials satisfying this constraint. Our results also apply in the case when the approximant is a rational fraction. Once $p$ has been found, one must prove that the error $|p - f|$ is not greater than a given bound. The particular form of the function $p - f$ (subtraction between two very close functions) makes this property hard to rigorously prove. We propose an algorithm for overcoming this difficulty.

All these algorithms have been integrated into a software tool called Sollya, developed during the thesis. In the beginning, it was created for making the implementation of functions easier. Now, it may be interesting for anyone who needs to perform numerical computations in a safe environment.

**Keywords:** polynomial approximation, floating-point arithmetic, numerical functions, rounding errors, supremum norm, euclidean lattices, software library, arbitrary precision.