Précision arbitraire : complexité, séries

AFAE 2020–2021

Marc Mezzarobba
marc@mezzarobba.net

Ces notes représentent le support de la séance 12 (à distance) du cours d'*Arithmétique flottante et analyse d'erreur* du master SFPN pour l'année universitaire 2020-2021. Elles contiennent très probablement des erreurs, utilisez-les à vos risques et périls. Les corrections sont les bienvenues.

Cours

1. Introduction

1.1. **Précision arbitraire.** Comme nous l'avons vu dans les cours précédents, une des façons d'améliorer la qualité des résultats d'un calcul numérique est simplement d'*augmenter la précision* des calculs intermédiaires. (Par exemple, en géométrie algorithmique, un problème classique est de décider *exactement* de quel côté d'une droite donnée par deux points à coordonnées `double` se trouve un troisième point.) Il y a aussi des situations, surtout en mathématiques et en physique théorique, où l'on est vraiment intéressé par un résultat avec des centaines, milliers, voire parfois millions de chiffres de précision.

Pour aller au-delà de la précision machine, on utilise des entiers ou des flottants à précision arbitraire implémentés en logiciel. Grossièrement, on peut distinguer deux approches principales de la grande précision :

1. La « petite grande précision », juste un peu au-delà de la précision des flottants machine (typiquement 100 à 300 bits). Celle-ci sert notamment à obtenir des résultats au format des flottants machine précis jusqu'au dernier bit malgré les erreurs d'arrondi dans les calculs intermédiaires. Elle peut être implémentée de plusieurs manières, notamment en représentant une valeur à grande précision comme somme non évaluée de flottants machine (« expansions flottantes »). La petite grande précision a été étudiée dans les cours des semaines 3, 5 et 10.
2. La précision arbitraire (d'une centaine à plusieurs millions de bits). C'est l'objet de ce cours-ci, et l'arithmétique que l'on trouve par exemple dans les logiciels de calcul formel comme Maple ou Sage. Les nombres sont codés en représentation multi-mots, c'est-à-dire comme tableaux d'entiers machine (« en base 2^{64} », par exemple). La précision des calculs n'est limitée que par la mémoire disponible. On cherche à rendre les opérations efficaces pour de grandes précisions.

1.2. **Grands entiers et grands flottants.** Plus précisément, dans ce cours, nous supposerons les entiers à précision arbitraire (aussi appelés *grands entiers*) représentés sous la forme

$$n = \pm \sum_{i=0}^{p-1} n_i \beta^i, \qquad 0 \leqslant n_i < \beta, \tag{1}$$

pour une certaine base β fixée. Nous prendrons $\beta = 2$ pour la théorie, mais $\beta = 2^{64}$ est plus proche de la réalité des implémentations.

De même, les flottants à précision arbitraire, ou grands flottants, seront représentés comme

$$x = m\,\beta^{e-p+1}, \qquad m, e \in \mathbb{Z}, \qquad \beta^{p-1} \leqslant |m| < \beta^p$$

où m et e sont des entiers. L'entier p s'appelle la précision de x. La mantisse m sera toujours un grand entier, mais pour simplifier les analyses de complexité nous ferons souvent l'hypothèse que l'exposant e est borné, ce qui revient à se limiter à des flottants x pris dans un intervalle borné fixé.

1.3. **Complexité.** Comme les nombres que l'on manipule en arithmétique à précision arbitraire peuvent être de taille (en mémoire) arbitrairement grande, une question théorique importante est celle de la *complexité* des opérations sur ces nombres.

Comme d'habitude en algorithmique, il s'agit d'estimer le temps de calcul d'un algorithme (le nombre d'opérations élémentaires qu'il demande) en fonction d'un ou plusieurs paramètres lorsque ces paramètres tendent vers l'infini. Un paramètre naturel ici est la *précision cible* (qui est aussi, au moins en première approximation, la taille de la sortie). Par exemple : quelle est la complexité, en fonction de p, de calculer π à précision 2^{-p} ?

Il faut aussi s'entendre sur les opérations élémentaires que l'on va compter : il n'est pas question de considérer l'addition ou la multiplication d'entiers quelconques comme une opération en temps constant[1] ! En revanche, les opérations arithmétiques sur les *chiffres* n_i de l'expression (1) prennent un temps $O(1)$.

Exercice 1. Le choix de la base β a-t-il une importance ? Pourquoi ?

Exercice 2. Quelle est la complexité de l'addition d'entiers d'au plus p chiffres ?

1.4. **Bibliographie.** Une bonne référence sur les points abordés dans ce cours est le livre *Modern Computer Arithmetic* de Richard P. Brent et Paul Zimmermann (Cambridge University Press, 2010), disponible en ligne ici :

https://members.loria.fr/PZimmermann/mca/pub226.html

2. Opérations arithmétiques de base

Considérons deux grands flottants

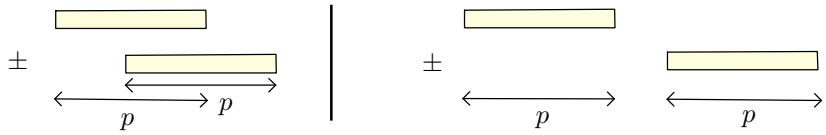
$$x = m_x\, 2^{e_x - p}, \qquad y = m_y\, 2^{e_y - p},$$

de même précision p, dont les exposants e_x, e_y tiennent sur au plus s chiffres. Pour chaque opération $\circledast \in \{+, -, \times, \ldots\}$, quelle est la complexité de calculer un arrondi fidèle à précision p de $x \circledast y$, par exemple sa valeur tronquée à p chiffres ?

Le but de cette section est d'établir des bornes de complexité en comprenant quels sont les phénomènes en jeu et non de donner des algorithmes optimisés ou même réalistes.

2.1. **Addition et soustraction.** Il y a plusieurs petites difficultés par rapport à l'addition d'entiers. Premièrement, x et y peuvent être d'ordres de grandeur très différents. Il faut « aligner » les chiffres de même poids avant d'additionner ou soustraire les mantisses, mais on ne peut pas se permettre de faire tout le calcul exactement puis tronquer le résultat à précision p : cela demanderait de travailler dans certains cas avec des entiers de taille $p + e_x - e_y$, qui peut être exponentiel en s. Deuxièmement, dans le cas d'une addition de nombre de signes opposés, un phénomène de compensation peut se produire.

1. En algorithmique des structures de données, on considère souvent (explicitement ou non) que les opérations sur les entiers se font en temps constant. C'est un modèle raisonnable quand les entiers en question sont, mettons, des indices de boucle ou de tableau, de taille (nombre de chiffres) beaucoup plus petite que la taille du problème, mais pas quand on cherche justement à analyser la complexité de tâches comme l'addition ou la multiplication d'entiers de taille arbitraire.



Néanmoins, il ne peut y avoir de compensation que si $e_x = e_y$: il suffit donc[2] toujours de faire le calcul « à précision p ». (Si les deux nombres ne se « recouvrent pas », le résultat est $\pm x$ ou $\pm y$, s'ils se « recouvrent complètement », il peut y avoir annulation mais on a fait le calcul exactement, s'ils se « recouvrent partiellement », il n'y a pas d'annulation.) Plus précisément, si par exemple $e_x \geqslant e_y$, on peut utiliser l'algorithme suivant :

1. Calculer $\delta = e_x - e_y$. $\mathrm{O}(s)$ ops
2. Calculer $m'_y = \lfloor m_y / 2^\delta \rfloor$ (décaler m_y de δ chiffres vers la droite, en oubliant les chiffres de poids faible). $\mathrm{O}(p)$ ops
3. Calculer $t = m_x \pm m'_y$ et son nombre de chiffres $0 \leqslant \ell \leqslant p+1$. $\mathrm{O}(p)$ ops
4. Décaler t de $p - \ell$ chiffres vers la gauche. $\mathrm{O}(p)$ ops
5. Calculer l'exposant final $e_x - (p - \ell)$ (si le résultat est $\neq 0$). $\mathrm{O}(s + \lg p)$ ops

Le coût total est $\mathrm{O}(p+s)$.

2.2. **Multiplication.** Il y a plusieurs algorithmes classiques pour multiplier deux entiers de taille au plus n — vous les avez sans doute déjà rencontrés, en cours d'algorithmique, de calcul formel ou de cryptographie.

- l'algorithme « naïf » (celui de l'école primaire) demande $\mathrm{O}(n^2)$ opérations ;
- l'algorithme de Karatsuba est une méthode diviser-pour-régner simple qui s'exécute en $\mathrm{O}(n^{\log_2 3}) = \mathrm{O}(n^{1.59})$ opérations ;
- toute une famille algorithmes utilisant la transformée de Fourier rapide (FFT) atteignent des complexités quasi-linéaires[3] en n.

La meilleure borne de complexité connue à ce jour, annoncée en mars 2019 par David Harvey et Joris van der Hoeven, provient d'une méthode sophistiquée à base de FFT. On conjecture que ce résultat est optimal. C'est une avancée théorique majeure — la question était ouverte depuis les années 1970 — mais sans impact pratique à ce stade.

Théorème 1. On peut calculer le produit de deux entiers de taille au plus n en $\mathrm{O}(n \log(n))$ opérations.

La multiplication rapide d'entiers est *la* brique de base essentielle pour aboutir à des algorithmes efficaces en arithmétique à précision arbitraire, tant en théorie qu'en pratique — ce qu'on résume par le slogan « *reduce everything to multiplication* ». Suivant le contexte (qualité de l'implémentation, taille des entrées...), son coût pratique peut aller de « plutôt quadratique » à « quasiment linéaire ». Il est d'usage de paramétrer les analyses de complexité par la complexité de la multiplication en utilisant la définition suivante.

2. Pour un arrondi *fidèle* ! obtenir l'arrondi correct est un peu plus délicat.

3. On dit que $f(n)$ est quasi-linéaire en $g(n)$, et l'on note $f(n) = \tilde{\mathrm{O}}(g(n))$, s'il existe des constantes c et k telles que, pour tout n, l'on ait $f(n) \leqslant c\, g(n) \log(g(n))^k$.

Définition 2. On appelle fonction de coût de la multiplication une fonction $\mathrm{M}: \mathbb{N}^* \to \mathbb{N}$ telle que :

1. il existe un algorithme de multiplication d'entiers qui, pour tout $n \geqslant 1$, calcule le produit de deux entiers de taille au plus n en au plus $\mathrm{M}(n)$ opérations ;
2. la fonction $n \mapsto \mathrm{M}(n)/n$ est croissante.

Exercice 3. Montrer que l'on a alors $\mathrm{M}(m) + \mathrm{M}(n) \leqslant \mathrm{M}(m+n)$ pour tous n et m. (Plus généralement, la deuxième hypothèse dans la définition est utile pour simplifier les bornes de complexité faisant intervenir la fonction M.)

La multiplication de grands flottants se ramène facilement à la multiplication d'entiers. On peut utiliser l'algorithme suivant :

1. Additionner les exposants. $\mathrm{O}(s)$
2. Mutliplier les mantisses. $\mathrm{O}(\mathrm{M}(p))$
3. Tronquer le produit à précision p. $\mathrm{O}(p)$

Le coût total est donc $\mathrm{O}(\mathrm{M}(p) + s)$. (On peut bien sûr ne calculer que les p premiers chiffres du produit des mantisses, mais cela n'améliore la complexité que d'un facteur constant.)

2.3. **Rappels sur la méthode de Newton.** L'outil de base pour calculer efficacement l'inverse d'un grand flottant est la méthode de Newton. Rappelons l'idée.

On a une fonction $f: \mathbb{R} \to \mathbb{R}$ (de classe $\mathcal{C}^2$), et l'on cherche un x tel que $f(x) = 0$. Supposons que l'on dispose déjà d'une approximation $\tilde{x} \approx x$. On écrit un développement limité à l'ordre 2 :

$$f(x) = f(\tilde{x}) + f'(\tilde{x})\,(x - \tilde{x}) + \mathrm{O}((x-\tilde{x})^2).$$

Comme $f(x) = 0$, il vient

$$x = \tilde{x} - \frac{f(\tilde{x})}{f'(\tilde{x})} + \mathrm{O}\left(\frac{(x-\tilde{x})^2}{f'(\tilde{x})}\right).$$

Si $\tilde{x} \approx x$, le facteur $(x - \tilde{x})^2$ qui apparaît dans le $\mathrm{O}(\cdot)$ est tout petit. Pourvu que $f'(\tilde{x})$ ne soit pas trop petit, on s'attend à ce que

$$N(\tilde{x}) := \tilde{x} - \frac{f(\tilde{x})}{f'(\tilde{x})}$$

soit une bien meilleure approximation de x que ne l'était $\tilde{x}$.

Exercice 4. Faire le dessin.

Cela suggère, à partir d'une approximation initiale x_0 obtenue par un autre moyen, de considérer l'itération

$$x_{n+1} = N(x_n).$$

Si tout se passe bien, on s'attend à ce que la suite (x_n) ainsi définie converge rapidement vers x.

2.4. **Inverse.** Détaillons maintenant le calcul de l'inverse. On suppose donné un grand flottant y comme ci-dessus, et l'on veut calculer $x = y^{-1}$. Quitte à multiplier y par une puissance de deux avant de commencer, on peut supposer

$$\frac{1}{2} \leqslant y < 1.$$

Pour tirer parti de la méthode de Newton, on va chercher x comme zéro d'une certaine fonction f. La première idée serait de prendre $f(x) = x\,y - 1$. L'itération de Newton s'écrit alors

$$x_{n+1} = x_n - \frac{x_n\,y - 1}{y} = \frac{1}{y}.$$

Il semblerait donc que pour calculer $1/y$, on ait déjà besoin de savoir calculer $1/y$…

Mais on peut essayer d'autres fonctions ! Prenons plutôt $f(x) = y - 1/x$. Alors $f'(x) = 1/x^2$, et on a l'itération

$$x_{n+1} = N(x_n) = x_n - \frac{y - 1/x_n}{1/x_n^2} = 2\,x_n - y\,x_n^2. \tag{2}$$

Cette fois, le passage de x_n à x_{n+1} ne demande que des additions et des multiplications, que l'on sait déjà faire ! Cette formule est donc prometteuse. Il reste à s'assurer que la suite x_n converge, et à comprendre à quelle vitesse.

Regardons déjà ce qu'il se passe si l'on fait le calcul en arithmétique exacte, sans erreurs d'arrondi. La proposition suivante dit qu'avec un choix convenable de condition initiale, la suite converge quadratiquement dès la première itération, autrement dit, le nombre de chiffres significatifs corrects du résultat double à chaque étape.

Proposition 3. Soit (x_n) la suite réelle définie par la récurrence (2) à partir de la condition initiale $x_0 = 3/2$. L'erreur relative

$$\varepsilon_n = \frac{x_n - y^{-1}}{y^{-1}}$$

vérifie $|\varepsilon_n| \leqslant 2^{-2^n}$ pour tout $n \in \mathbb{N}$.

Exercice 5. Prouver la proposition.

En réalité, cependant, on travaille avec des approximations numériques des x_n. Pour évaluer la complexité de l'algorithme d'inversion, il nous faut préciser à quelle précision les calculs intermédiaires seront effectués. Le coût des additions et multiplications à chaque itération en dépend.

On vient de voir que dans le calcul exact, x_n est une approximation de ξ_n avec environ 2^n bits corrects. De plus, la preuve de la proposition montre en fait que pour *toute* approximation ξ_n de y^{-1} avec $\approx 2^n$ bits corrects (c.-à-d. avec $|\xi_n\,y - 1| \lesssim 2^{-2^n}$), $N(\xi_n)$ est une approximation de y^{-1} avec $\approx 2^{n+1}$ bits corrects. Perturber un peu x_n ne devrait donc pas être un problème, et ce serait du gâchis de le représenter à une précision beaucoup plus grande que 2^n.

On considère donc l'itération approchée suivante, où $\mathrm{rnd}(a, p)$ est n'importe quel arrondi fidèle de a à précision p :

$$x_{n+1} = \mathrm{rnd}(2\,x_n - y\,x_n^2, 2^{n+1}). \tag{3}$$

On admet la borne d'erreur suivante. (La preuve est du même tonneau que la précédente, mais plus compliquée. C'est un exercice un peu difficile… à ne faire que s'il vous reste du temps !)

Lemme 4. Soit (x_n) la suite réelle définie par l'itération approchée (3) à partir de la condition initiale $x_0 = 3/2$. On a $|x_n\,y - 1| \leqslant 2^{-2^{n-1}-1}$ pour tout $n \geqslant 1$.

On peut maintenant étudier la complexité de l'algorithme.

Proposition 5. Étant donné $y \in [1/2, 1[$, l'itération de Newton avec arrondi (3) calcule y^{-1} avec une erreur relative bornée par 2^{-p} en $\mathrm{O}(\mathrm{M}(p))$ opérations.

Démonstration. Commençons par voir au bout de combien d'itérations on peut s'arrêter. D'après le lemme, on a $|x_n\, y - 1| \leqslant 2^{-2^{n-1}-1}$ pour $n \geqslant 1$. Pour rendre cette borne plus petite que 2^{-p}, il suffit de prendre $n = N = \lceil \log_2 p \rceil + 2$. On a alors $2^N = \mathrm{O}(p)$.

Comme l'itération n ne calcule x_n qu'à précision 2^n, son coût est $\leqslant c\,\mathrm{M}(2^n)$ pour une certaine constante c, qui ne dépend pas de n. Le coût total s'obtient en sommant les coûts des itérations, ici à partir de la dernière :

$$\begin{aligned} c\,\mathrm{M}(2^N) + c\,\mathrm{M}(2^{N-1}) + \cdots + c\,\mathrm{M}(2^0) &\leqslant c\,\mathrm{M}(2^N + 2^{N-1} + \cdots + 2^0) \\ &\leqslant c\,\mathrm{M}(2^{N+1}) \\ &= \mathrm{O}(\mathrm{M}(p)). \end{aligned}$$

Remarquons que c'est mieux que la borne $\mathrm{O}(\mathrm{M}(p)\log(p))$ que l'on obtiendrait en faisant tout le calcul à la précision finale p. □

3. Évaluation de fonctions

3.1. **Précision fixée *vs* précision arbitraire.** On s'intéresse maintenant à la complexité de l'évaluation de fonctions mathématiques. Le problème est le suivant : pour une fonction $f : \mathbb{R} \to \mathbb{R}$ (ou $f : \mathbb{C} \to \mathbb{C}$) fixée, développer un algorithme rapide qui prend en entrée un point x et une précision p, et renvoie une approximation de $f(x)$ avec (environ) p bits corrects.

Quand la précision est fixée (cf. cours 9), on commence quand c'est possible par appliquer une *réduction d'argument* qui dépend de la fonction pour se ramener à un ou plusieurs petits domaines. Sur ces petits domaines, on remplace f par des approximations précalculées, par exemple des polynômes.

Le problème à précision arbitraire est bien sûr que l'on ne peut pas *précalculer* les approximations, puisque l'on ne connaît pas à l'avance la précision demandée. On choisit donc des approximations suffisamment « structurées » pour pouvoir les calculer efficacement au vol, par exemple des séries tronquées.

3.2. **Stratégie générale, exemples.**

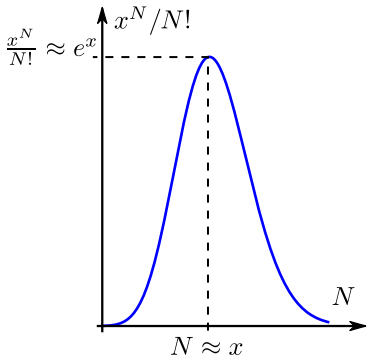


Figure 1. Allure des termes du développement en série de $\exp(x)$ pour $x > 1$.

Exemple 6. Exponentielle.

1. *Réduction d'argument.* Appliquer les formules $e^{-x} = 1/e^x$ et $e^x = (e^{x/2})^2$ pour se ramener à $x \geqslant 0$ « petit » (par exemple $x \in [0, 1/2]$).

2. *Sommation d'une série.*

$$e^x = \underbrace{\sum_{n=0}^{N-1} \frac{x^n}{n!}}_{\text{approximation}} + \underbrace{\frac{x^N}{N!} \sum_{n=0}^{\infty} \frac{x^n}{(N+1)\,(N+2)\cdots(N+n)}}_{\text{reste (erreur)}}.$$

→ Les coefficients se calculent facilement et efficacement par récurrence :

$$\frac{1}{(n+1)!} = \frac{1}{n+1} \cdot \frac{1}{n!}.$$

→ Pour $x \geqslant 0$, on a $0 \leqslant \sum_{n=0}^{\infty} \frac{x^n}{(N+1)\,(N+2)\cdots(N+n)} \leqslant e^x$. L'erreur relative quand on tronque la série après N termes est donc bornée par $x^N / N!$. Pour rendre cette borne $\leqslant 2^{-p}$, il suffit de prendre

$$N \approx \frac{p}{\ln(p/x)}.$$

→ Remarque : la série converge pour tout $x \in \mathbb{R}$. Mais plus x est grand, plus elle « commence à converger tard » (voir figure 1) et plus la convergence est lente. D'où l'intérêt d'avoir tout de même une phase de réduction d'argument !

3. *Reconstruction.* Carrés et inverses pour retrouver e^x à partir de l'exponentielle de l'argument réduit.

Exemple 7. La fonction d'erreur, très utilisée en probabilités et statistiques, est définie par

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,\mathrm{d}t = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{2n+1}}{n!\,(2\,n+1)}. \tag{4}$$

Son évaluation est plus compliquée que l'exponentielle faute d'une réduction d'argument intéressante.

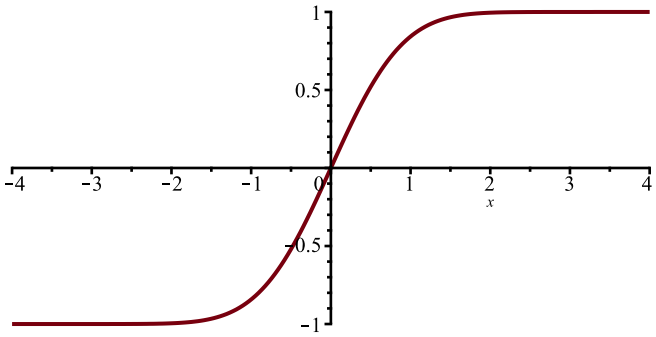


Figure 2. $\operatorname{erf}(x)$

1. *Réduction d'argument.* On peut se ramener à $x \geqslant 0$ car $\operatorname{erf}(-x) = \operatorname{erf}(x)$, mais il n'y a pas de relation algébrique simple pour se ramener à x petit. On va donc se contenter de découper le domaine.
2. *Développement en série en zéro.* Comme pour l'exponentielle, (4) est un développement explicite en série convergente au voisinage de zéro. Comme pour l'exponentielle, ce développement est utilisable pour x assez petit, mais présente une « bosse » (figure 1) très haute et converge lentement quand x est grand.

Difficulté supplémentaire : la série est alternée, et le n-ième terme est de l'ordre de $x^{2n}/n!$. Il est maximal (en valeur absolue) pour $n \approx x^2$, en vaut alors $\approx \exp(x^2)$. Mais on a $\operatorname{erf}(x) \to 1$ quand $x \to +\infty$. Cela signifie que si l'on utilise la série pour calculer $\operatorname{erf}(x)$ pour x grand, la valeur (proche de 1) est obtenue en ajoutant et soustrayant des termes de l'ordre de $\exp(x^2)$.

Un problème d'*annulation catastrophique* dans l'évaluation de la série s'ajoute donc à la lenteur de convergence. Si l'on veut tout de même utiliser cette série pour x grand, il faut faire les calculs intermédiaires à une précision considérablement plus grande que celle du résultat.

3. *Développement asymptotique à l'infini.* On peut montrer que pour tout N,

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

quand $x \to \infty$. Les coefficients se calculent facilement par récurrence.

Attention : la série est *divergente* quand x est fixé, aussi grand qu'il soit ! Malgré cela, pour n'importe quel N fixé, la quantité

$$s(x, N) = 1 + \frac{e^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{N-1} (-1)^n \frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2^n} \frac{1}{x^{2n+1}} \tag{5}$$

se rapproche d'autant plus de $\operatorname{erf}(x)$ que x est grand, et il est possible d'obtenir des bornes sur la différence. On peut donc se servir de cette formule avec un N judicieux *si la précision demandée est assez petite* (voir figure 3).

Dit encore autrement : à x et ε donnés, il est possible qu'il n'existe pas de N tel que $|\operatorname{erf}(x) - s(x, N)| \leqslant \varepsilon$. Mais à ε fixé, il existe un tel N pour tout x suffisamment grand. La précision ε jusqu'à laquelle la série divergente est utilisable dépend de x : plus x est grand, plus on peut l'utiliser pour ε petit.

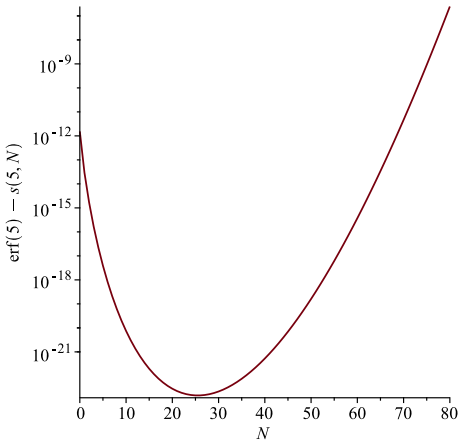


Figure 3. L'erreur d'approximation de $\operatorname{erf}(x)$ par $s(x, N)$ pour $x = 5$ commence par décroître pour $N \leqslant 25$, puis tend vers l'infini. Elle est minimale en $N = 25$, où elle vaut environ $5.5 \cdot 10^{-23}$. C'est la meilleure précision que l'on peut obtenir avec cette série ; pour aller au-delà, il faut utiliser une autre méthode. Pour x plus grand, la forme générale de la courbe est semblable, mais les termes décroissent pendant plus longtemps et l'erreur minimale est plus petite.

Un algorithme simple d'évaluation de $\mathrm{erf}(x)$ à précision p est donc :

- si x est assez grand pour qu'il existe N tel que la formule (5) donne une approximation à précision p, calculer la somme partielle correspondante, en déduire $\mathrm{erf}(x)$;
- sinon, utiliser le développement en série en zéro (avec éventuellement des astuces qui limitent la perte de précision due à l'annulation catastrophique).

Ce schéma est assez général : souvent, l'évaluation à précision arbitraire d'une fonction se ramène après quelques transformations à calculer une ou plusieurs sommes partielles de séries. Pour de nombreuses fonctions usuelles, on combine une série convergente en zéro et un développement asymptotique divergent à l'infini. Les sommes partielles de l'une et de l'autre sont souvent de la forme

$$\varphi(x) \sum_{n=0}^{N} a_n \, x^n$$

où $\varphi(x)$ est une fonction auxiliaire simple et où les coefficients a_n sont donnés par

$$a_{n+1} = \frac{g(n)}{h(n)} \, a_n \tag{6}$$

avec g et h polynômes. Une série dont les coefficients satisfont une récurrence de cette forme s'appelle une série *hypergéométrique*.

On va donc se pencher sur le calcul efficace de sommes de ce type.

3.3. **Sommation directe.** Intéressons-nous au cas d'une série convergente[4]. Soit à calculer à précision p la somme

$$S = \sum_{n=0}^{\infty} a_n \, x^n$$

où les a_n vérifient (6).

Pour simplifier l'analyse, supposons $1 \leqslant S < 2$, de sorte qu'il revient au même de calculer S à précision relative p ou à précision absolue p. Supposons de plus que les a_n vérifient $|a_n| \leqslant \alpha^n$ pour n assez grand, et que l'on a $|x| < 1 \,/\, \alpha$ (le point d'évaluation est donc à l'intérieur du disque de convergence de la série). En pratique, on se ramène souvent à cette situation.

L'algorithme évident pour calculer une somme partielle est le suivant.

Algorithme 8. SommeHypergéométrique(N, p)

1. $S := 0$; $\quad t := a_0$
2. Pour n de 0 à $N-1$
   a. $S := S + t$ $\quad$ (à précision p)
   b. $t := \frac{g(n)}{h(n)} \, x \, t$ $\quad$ (à précision p)
3. Renvoyer S

Proposition 9. L'algorithme 8 s'exécute en temps $\mathrm{O}(N\,(\mathrm{M}(p) + \log(N)))$.

Démonstration. On admet que les exposants des flottants manipulés par l'algorithme sont de taille $\leqslant s = c \cdot \log(N)$ pour une certaine constante c. (Ce n'est pas très dur à vérifier par récurrence, au moins si on néglige les erreurs d'arrondi.)

4. Dans le cas d'un développement asymptotique divergent, l'analyse est plus compliquée et plus *ad hoc*, mais les conclusions de cette section restent souvent vraies en pratique.

D'après la section 2.1, l'étape 2a prend un temps $O(p+s)$. Les polynômes g et h sont fixés, donc leur évaluation en n demande un nombre borné d'additions et de multiplications. Le coût de l'étape 2b est donc $O(M(p)+s)$. Les constantes cachées par ces deux $O(\cdot)$ sont indépendantes de n, et $p = O(M(p))$ par définition d'une fonction de multiplication, donc le coût total de l'algorithme est

$$O(N\,(M(p)+p+s)) = O(N\,(M(p)+\log(N))).$$ □

Maintenant, de quelle valeur de N a-t-on besoin ?

Lemme 10. Pour calculer S à précision p, il suffit de sommer $N = O(p)$ premiers termes.

Démonstration. Le reste de la série tronquée à N termes s'écrit

$$R_N = \sum_{n \geqslant N} a_n\, x^n.$$

On a donc

$$|R_N| \leqslant \sum_{n \geqslant N} \alpha^n\, |x|^n = \frac{\alpha^N\, |x|^N}{1-\alpha\,|x|}.$$

Pour x fixé, comme $\alpha\,|x| < 1$, cette borne devient plus petite que 2^{-p} pour $N = O(p)$ quand $p \to \infty$. □

Bilan : s'il n'y a pas trop d'erreurs numériques dans le calcul de la somme partielle, l'algorithme 8 appelé avec les paramètres $(K \cdot p, L \cdot p)$ où K, L dépendent de la série mais pas de p fournit une approximation de S à précision p en

$$O(p\,M(p))$$

opérations. Cette complexité est *grosso modo* quadratique en la précision. Peut-on faire mieux ?

3.4. **Arbres de produits.** Nous allons voir que la réponse est oui, au moins quand le point d'évaluation x est un rationnel fixé indépendant de la précision d'évaluation, plutôt qu'un flottant de précision p. Le point de départ est la notion d'arbre de produits, que vous auriez déjà pu rencontrer, par exemple, dans le contexte de l'évaluation-interpolation rapide de polynômes.

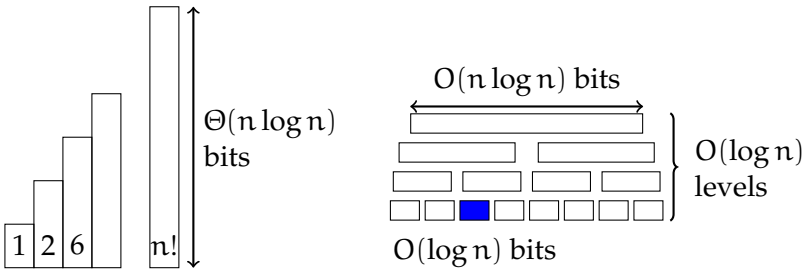


Figure 4. Calcul direct de la factorielle *vs* calcul par scindage binaire

Commençons par un exemple simple. L'algorithme évident pour calculer la factorielle d'un entier n est de former le produit

$$n! = (((1 \times 2) \times 3) \times \cdots) \times n.$$

Comme $k!$ est un entier de $k\ln(k)$ bits environ, la complexité de cet algorithme est de l'ordre de

$$\mathrm{M}(n\ln(n)) + \mathrm{M}((n-1)\ln(n-1)) + \cdots$$

opérations. Oublions même les $\mathrm{M}(\cdot)$ et les $\ln(k)$: la taille totale des résultats intermédiaires que calcule cet algorithme (partie gauche de la figure 4) est d'au moins

$$n + (n-1) + (n-2) + \cdots + 1 \sim \frac{n^2}{2}$$

bits, et comme il faut bien écrire en mémoire chacun des résultats intermédiaires, la complexité est $\Omega(n^2)$.

Une autre manière classique de calculer $n!$ consiste à multiplier d'abord les entiers $1, ..., n$ deux à deux, puis multiplier les résultats obtenus deux à deux, et ainsi de suite jusqu'à n'avoir plus qu'un entier. L'intérêt de cette méthode est que le produits sont plus équilibrés (partie droite de la figure 4) : on multiplie des entiers qui font à peu près la même taille plutôt que de très grands entiers par des petits, ce qui permet de mieux profiter de l'efficacité de la multiplication rapide.

Plus formellement, posons

$$P(i,j) = (i+1)\cdots j.$$

On a donc $n! = P(1,n)$. L'algorithme consiste à calculer récursivement ce produit en utilisant la formule

$$P(i,j) = P(m,j)\cdot P(i,m), \qquad m = \left\lfloor \frac{i+j}{2} \right\rfloor.$$

Cet algorithme est ausi appelé méthode par *scindage binaire*.

Si $n = 2^k$, le coût du calcul est borné (à une constante près) par

$$C(n) = \mathrm{M}\big(\tfrac{n}{2}\log n\big) + 2\,\mathrm{M}\big(\tfrac{n}{4}\log n\big) + \cdots + 2^{k-1}\,\mathrm{M}(\log n).$$

D'après l'exercice 3, on a

$$\begin{aligned} C(n) &\leqslant k\,\mathrm{M}(2^k \log n) \\ &= \mathrm{O}(\mathrm{M}(n\log n)\log n). \end{aligned}$$

Avec une multiplication quasi-linéaire, cette complexité est quasi-linéaire.

La même idée fonctionne si l'on remplace $n!$ par un produit

$$g(1)\,g(2)\cdots g(n)$$

pour un polyôme $g \in \mathbb{Z}[X]$ fixé, ou même par un produit de matrices à coefficients entiers. On aboutit à la proposition suivante.

Proposition 11. Soit $A \in \mathbb{Z}[X]^{r\times r}$ une matrice $r \times r$ de polynômes à coefficients entiers fixée. On peut calculer le produit

$$A(n)\,A(n-1)\cdots A(1)$$

en $\mathrm{O}(\mathrm{M}(n\log n)\log n)$ opérations binaires.

> **Exercice 6.** Compléter les détails de la preuve : pourquoi l'analyse de complexité réalisée pour la factorielle est-elle encore valable ?

3.5. **Sommation de séries par scindage binaire.** Notre but est maintenant d'utiliser ce résultat pour donner un algorithme qui évalue à précision p la somme d'une série hypergéométrique convergente en temps quasi-linéaire en p.

Comme ci-dessus, on considère donc

$$S = \sum_{n=0}^{\infty} a_n\, x^n$$

où les a_n vérifient (6), avec de plus l'hypothèse que x est un « petit » rationnel (non pas au sens ou sa valeur absolue est petite mais au sens où il s'écrit avec peu de chiffres). Ces hypothèses couvrent par exemple le calcul de $\exp(3)$ ou de $\mathrm{erf}(1/2)$.

Notons $t_n = a_n\, x^n$ et

$$S_N = \sum_{n=0}^{N-1} t_n.$$

On a donc

$$\begin{aligned} t_{n+1} &= \left(\frac{g(n)}{h(n)}\, x\right) t_n, \\ S_{n+1} &= S_n + t_n. \end{aligned}$$

C'est un système de deux récurrences qui peut se réécrire sous forme matricielle

$$\begin{bmatrix} t_{n+1} \\ S_{n+1} \end{bmatrix} = \begin{bmatrix} \frac{g(n)}{h(n)}\, x & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} t_n \\ S_n \end{bmatrix} = \frac{1}{s\, h(n)}\, A(n) \begin{bmatrix} t_n \\ S_n \end{bmatrix} \tag{7}$$

avec $x = r/s$ où $r, s \in \mathbb{Z}$ et

$$A(n) = \begin{bmatrix} r\, g(n) & 0 \\ s\, h(n) & s\, h(n) \end{bmatrix}.$$

Algorithme 12. ScindageBinaireHypergéométrique(N, p)

1. Calculer
$$U = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} = A(N-1) \cdots A(1)\, A(0)$$
en formant un arbre de produits.
2. Calculer $S_N = a_0\, u_{21}/u_{22}$ en faisant la multiplication exactement puis la division à précision p.
3. Renvoyer S_N.

Proposition 13. L'algorithme 12 calcule un arrondi à précision p de S_N en

$$\mathrm{O}\Big(\mathrm{M}\big(N \log(N)\big) \log(N) + \mathrm{M}(p)\Big)$$

opérations.

Démonstration. D'après (7), on a

$$\begin{bmatrix} t_N \\ S_N \end{bmatrix} = \frac{1}{s^n\, h(N-1)\, h(N-2) \cdots h(0)}\, A(N-1)\, A(N-2) \cdots A(0).$$

On remarque de plus que le coefficient u_{22} du produit $U = A(N-1) \cdots A(1)\, A(0)$ est égal au dénominateur $s^N\, h(N-1)\, h(N-2) \cdots h(0)$. Par conséquent, la formule $S_N = a_0\, u_{21}/u_{22}$ est une expression exacte de S_N. Son calcul en faisant la division à précision p donne donc un arrondi à précision p de S_N.

La complexité de la première étape est $\mathrm{O}(\mathrm{M}(N \log N) \log N)$ d'après la section précédente, et celle de la seconde étape est $\mathrm{O}(\mathrm{M}(p))$. □

Dans les hypothèses considérées plus haut où $N = \mathrm{O}(p)$ convient pour avoir une approximation à précision p, le coût total de l'algorithme est donc

$$\mathrm{O}\big(\mathrm{M}\big(p\log(p)\big)\log(p)\big).$$

Cette complexité est quasi-linéaire en p. En particulier, on peut calculer e, π, et toutes sortes d'autres constantes usuelles à précision p dans cette complexité. Le TME qui suit propose d'implémenter l'algorithme étudié ici dans le cas de π.

TME : Calcul de π à grande précision

Le but de ce TME est d'écrire un programme capable de calculer un million de décimales de π en quelques secondes, en utilisant la bibliothèque de calcul en précision arbitraire GNU MP.

Nous partirons de la formule suivante, due à D. et G. Chudnovsky en 1988, qui fournit une expression de π en fonction d'une série à convergence très rapide :

$$\frac{1}{\pi} = \frac{12}{c^{3/2}} \sum_{n=0}^{\infty} t_n \qquad \text{avec} \qquad t_n = (-1)^n \frac{(6n)!}{(3\,n)!\,n!^3} \frac{(a\,n+b)}{c^{3n}}, \tag{8}$$

où

$$a = 545140134, \qquad b = 13591409, \qquad c = 640320.$$

Un squelette de code à compléter est disponible à l'adresse

http://marc.mezzarobba.net/enseignement/2021-afae/pi.c

et un corrigé à l'adresse

https://gitlab.lip6.fr/mezzarobba/afae2021-tme-pi

1. Que calcule la fonction `f` ci-dessous ? En s'aidant du manuel de GMP (accessible à l'adresse `https://gmplib.org/manual/`), comprendre le rôle de chaque intruction, puis adapter la définition de `f` pour lui faire calculer $a\,n+b$.

```c
#include <gmp.h>
void f(mpz_t res, unsigned long n)
{
    mpz_t a, b;
    mpz_init_set_ui(a, 545140134);
    mpz_init_set_ui(b, 13591409);
    mpz_mul(res, a, b);
    mpz_add_ui(res, res, n);
    mpz_clear(b);
    mpz_clear(a);
}
```

```c
int main(void) {
    mpz_t x;
    mpz_init(x);
    f(x, 42);
    gmp_printf("%Zd\n", x);
    mpz_clear(x);
}
```

2. Exprimer t_{n+1}/t_n comme une fraction rationnelle en n.
3. Écrire une fonction de prototype

```c
void term_ratio (mpz_t num, mpz_t den, unsigned long n)
```

qui calcule deux entiers `num` et `den` premiers entre eux tels que `num` / `den` $= t_{n+1}/t_n$. Afficher les valeurs correspondantes pour $0 \leqslant n \leqslant 9$. Par exemple,

$$t_4/t_3 = -58364441737/100388937802668320489472000.$$

On pourra supposer que $6\,n+6$ est représentable par un entier de type `unsigned long`.

4. Soit $s_n = \sum_{k=0}^{n-1} t_k$ la n-ième somme partielle de la série apparaissant dans (8). En utilisant la question 2, déterminer un polynôme $q(X)$ et une matrice de polynômes $A(X)$ tels que

$$\begin{pmatrix} t_{n+1} \\ s_{n+1} \end{pmatrix} = \frac{1}{q(n)} A(n) \begin{pmatrix} t_n \\ s_n \end{pmatrix}.$$

Exprimer s_n à partir des coefficients de la matrice produit

$$A(n-1)\,A(n-2)\cdots A(1)\,A(0).$$

5. On introduit le type[5]

```
typedef struct { mpz_t t, s, q; } mat[1];
```

pour représenter les matrices de la forme $\begin{pmatrix} T & 0 \\ S & Q \end{pmatrix}$.

Écrire des fonctions

```
void mat_init (mat m);
void mat_clear (mat m);
void mat_mul (mat res, mat high, mat low);
```

pour, respectivement, initialiser, désallouer et multiplier les objets de type `mat`. On suivra les conventions de gestion de la mémoire et de passage des paramètres de GMP.

6. Écrire une fonction

```
void binsplit (mat res, unsigned long i, unsigned long j);
```

qui calcule récursivement le produit

$$\begin{pmatrix} T_{i,j} & 0 \\ S_{i,j} & Q_{i,j} \end{pmatrix} = A(j-1)\cdots A(i+1)\,A(i),$$

par scindage binaire. Vérifier que l'on a

$$\begin{aligned} S_{0,1}/Q_{0,1} &= 1,\\ S_{0,2}/Q_{0,2} &= 29735444608353174286057/29735444608353733017600,\\ S_{0,11}/Q_{0,11} &= 84667...82283/84667...00000. \end{aligned}$$

7. Estimer à quel ordre N tronquer la série apparaissant dans (8) pour calculer sa somme à 10^{-p} près, quand p tend vers l'infini.

8. En déduire une fonction

```
void pi (mpf_t res, unsigned long prec);
```

qui calcule une approximation de π avec environ `prec` décimales correctes.

Celle-ci calculera s_N sous la forme d'un quotient *non simplifié* de deux entiers à l'aide de la fonction `binsplit` de la question 6. Pour en déduire une approximation de π à l'aide de l'équation (8), on pourra utiliser les fonctions d'arithmétique *flottante* à précision arbitraire de GMP ou de MPFR.

5. Un objet de type `mat` est donc un *tableau à un élément* de structures. L'intérêt de cette définition un peu étrange est qu'ainsi, déclarer une variable de type `mat` alloue la mémoire correspondante, mais appeler une fonction qui prend un objet de type `mat` en paramètre passe un *pointeur* vers l'argument. Autrement dit, les objets de type `mat` sont automatiquement passés par référence. GMP utilise la même astuce pour les types `mpz_t`, `mpq_t`, etc. C'est ce qui permet à la fonction `f` de la question 1 de renvoyer une valeur « dans » l'objet `res` qui lui est donné en paramètre sans demander de passer un pointeur.

On donne pour aider à débuguer

$$\begin{aligned}\frac{c^{3/2}}{12\,\pi} &\approx 13591408.999999744616264456976 0,\\ \frac{12\,\pi}{c^{3/2}} &\approx 7.3575889004592444395233775374\cdot 10^{-8},\end{aligned}$$

et quelques décimales de π à partir des positions 1, 10, 100, 1000, ... après la virgule :

$\pi = 3.$**141592**...**589793**...**982148**...**938095**...**856672**...**641260**...**130927**...

Combien de décimales arrivez-vous à calculer en une seconde ? En une minute ?