Département de formation doctorale en informatique
UFR STMIA

École doctorale IAEM Lorraine

De l'algorithmique à l'arithmétique via le calcul formel

MÉMOIRE

présenté et soutenu publiquement le 26 novembre 2001

pour l'obtention de l'

Habilitation à diriger des recherches
(Spécialité informatique)

par

Paul ZIMMERMANN

Composition du jury

Rapporteurs :	Richard Brent
	Didier Galmiche
	Jean Vuillemin
Examinateurs :	Jean-Michel Muller
	Joël Rivat

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503
Institut National de Recherche en Informatique et en Automatique

Mis en page avec la classe thloria.

À Marie,
Louis et Iris,

Résumé

Ce mémoire présente mes travaux de recherche de 1988 à 2001, travaux effectués d'abord à l'INRIA Rocquencourt au sein du projet Algo (1988 à 1992), puis à l'INRIA Lorraine et au LORIA dans les projets Euréca (1993 à 1997), PolKA (1998 à 2000), et Spaces (2001). Au niveau thématique, on peut distinguer grosso modo trois phases : une première période allant de 1988 à 1992 où j'ai surtout travaillé sur l'analyse d'algorithmes et la génération aléatoire, une seconde période de 1993 à 1997 où je me suis investi dans le calcul formel et les algorithmes sous-jacents, enfin une troisième période depuis 1998 où je me suis intéressé aux problèmes d'arithmétique exacte en précision arbitraire.

Abstract

This document presents my research contributions from 1988 to 2001, performed first at INRIA Rocquencourt within the Algo project (1988 to 1992), then at INRIA Lorraine and LORIA within the projects Euréca (1993-1997), PolKA (1998-2000), and Spaces (2001). Three main periods can be roughly distinguished : from 1988 to 1992 where my research focused on analysis of algorithms and random generation, from 1993 to 1997 where I worked on computer algebra and related algorithms, finally from 1998 to 2001 where I was interested in arbitrary precision floating-point arithmetic with well-defined semantics.

Remerciements

Je tiens à remercier tout d'abord Philippe Flajolet et Benno Fuchssteiner, qui m'ont initié respectivement aux joies de l'analyse d'algorithmes et au développement d'un système de calcul formel.

La plupart des résultats décrits ici n'auraient pas été obtenus sans l'aide de plusieurs coauteurs, et nombre de mes travaux ont été motivés par des discussions avec mes « compères » de recherche Bruno Salvy, Fabrice Rouillier et Guillaume Hanrot. Un grand merci aussi à Vincent Lefèvre pour sa lecture attentive d'une première version de ce document.

Enfin merci à Richard Brent, Jean Vuillemin, Jean-Michel Muller, Joël Rivat et Didier Galmiche, qui m'ont fait l'honneur de faire partie du jury. Richard Brent est en quelque sorte mon « père spirituel » : dans chaque nouveau domaine de recherche que j'aborde, je ne tarde pas à m'apercevoir qu'il a apporté des contributions significatives, souvent très difficiles à améliorer ! Jean Vuillemin a été l'instigateur de mon intérêt pour l'algorithmique des grands nombres et la réalisation de logiciel en équipe, puisque le premier auquel j'ai contribué fut BigNum, bibliothèque développée en commun par l'INRIA et Digital PRL. Jean-Michel Muller m'a initié aux secrets de l'arithmétique exacte, et à l'arithmétique des ordinateurs, source inépuisable de problèmes aussi intéressants que difficiles. Avec Joël Rivat, nous partageons un intérêt pour les « mathématiques expérimentales », les implantations efficaces et les records en tous genres. Enfin, sans les encouragements répétés de Didier Galmiche au cours de la longue rédaction, ce mémoire n'aurait sans doute jamais vu le jour.

Table des matières

Introduction

Ce document présente mes travaux en vue de l'obtention de l'habilitation à diriger des recherches. Cette introduction résume mes contributions en les regroupant par thèmes. On peut distinguer grosso modo trois phases : une première période allant de 1988 à 1992 où j'ai surtout travaillé sur l'analyse d'algorithmes et la génération aléatoire, une seconde période de 1993 à 1997 où je me suis investi dans le calcul formel et les algorithmes sous-jacents, enfin une troisième période depuis 1998 où je m'intéresse plus aux problèmes d'arithmétique exacte en précision arbitraire.

Un cheminement naturel m'a mené de l'un à l'autre de ces trois axes de recherche. Les travaux que j'ai effectués en analyse d'algorithmes étaient validés par une implantation utilisant un logiciel de calcul formel, en l'occurrence Maple, ce qui m'a permis de me familiariser avec ce genre de logiciel. La suite logique de ces travaux en génération aléatoire fut également implantée en Maple. Cela m'a amené tout naturellement à m'intéresser de plus près au calcul formel, à la fois en tant qu'utilisateur et en tant que développeur. L'utilisation intensive de logiciels de calcul formel m'a permis de constater que ceux-ci étaient loin d'être parfaits pour les calculs numériques en précision arbitraire. Cela m'a poussé à m'interroger sur les moyens possibles pour remédier à cet état de fait, et m'a conduit logiquement à regarder les normes existant en précision fixe, d'où mon intérêt pour l'arithmétique exacte.

Le document est organisé de la façon suivante : mes travaux en algorithmique sont décrits dans le chapitre 1, ceux en calcul formel dans le chapitre 2, ceux en arithmétique dans le chapitre 3, puis le chapitre 4 rassemble quelques « calculs de l'extrême » auxquels j'ai participé, enfin quelques perspectives de ces travaux sont présentées.

Chapitre 1

Algorithmique

Initiées au contact de Philippe Flajolet dans le domaine de l'analyse en moyenne (§1.1), mes recherches en algorithmique se sont naturellement orientées vers la génération aléatoire (§1.2), sujet qui a également donné lieu à des contributions en calcul formel, mentionnées en §2. La thèse de François Bertault,[1] que j'ai encadrée, complète les volets « analyse » et « génération aléatoire » par un troisième volet sur le tracé de structures décomposables. Certains travaux ponctuels, comme le logiciel EPELLE de détection de fautes d'orthographe (§1.3) ou plus récemment l'étude de certains chemins du plan discret (§1.4), ne sont pas dans cette ligne directrice, mais ont bénéficié néanmoins des structures de données ou méthodes étudiées.

1.1 Analyse en moyenne : Lambda-Upsilon-Omega

L'analyse en moyenne d'algorithmes était le sujet de ma thèse de doctorat [25], et du logiciel Lambda-Upsilon-Omega réalisé en commun avec Bruno Salvy [12, 13]. L'objectif était de valider et d'étendre les travaux théoriques entrepris par Philippe Flajolet et Jean-Marc Steyaert quelques années auparavant.[2] Ce travail m'a permis notamment de me familiariser avec les outils puissants que sont les séries génératrices, l'analyse complexe et le calcul formel.

Un des résultats principaux de ma thèse est le théorème suivant sur l'analyse algébrique d'une classe de programmes opérant sur une classe de structures récursives non étiquetées (Ω) :

Théorème 1 *[25, Théorème 4] Tout programme dont la spécification des données est dans Ω, et dont les procédures sont définies à partir d'instructions élémentaires, des schémas d'exécution séquentielle, de sélection dans une union ou dans un produit cartésien, de sélection ou d'itération dans une séquence, un ensemble, un cycle ou un multi-ensemble, se traduit en un système d'équations pour les descripteurs de complexité associés aux procédures. Ces équations sont de la forme $\tau = f$, $\tau = \tau' + \tau''$, $\tau = f\tau'$, $\tau = \Theta(\tau', f)$ où*

[1] F. Bertault, *Génération et tracé de structures décomposables*, thèse de l'université Henri Poincaré Nancy 1, 1997.

[2] Ph. Flajolet, J.-M. Steyaert, *A Complexity Calculus for Recursive Tree Algorithms*, Mathematical Systems Theory 19, 1987.

τ, τ' et τ'' sont des descripteurs de complexité, f une fonction de la classe $\mathcal{UR}$ et Θ l'un des opérateurs suivants [...]

(La fin du théorème précise la forme des équations obtenues pour cette classe.)

Ce théorème a été implanté dans le logiciel Lambda-Upsilon-Omega co-écrit avec Bruno Salvy, avec d'autres résultats similaires. L'une des originalités de Lambda-Upsilon-Omega est d'appliquer à des programmes la méthode récursive bien connue des combinatoriciens pour l'énumération de structures de données (Schutzenberger, Foata, Viennot, Labelle, Leroux). Les programmes sont donc eux aussi considérés comme des structures de données, la récursivité permettant d'analyser des fonctions récursives voire mutuellement récursives, ce que d'autres méthodes d'analyse ne savent pas bien faire. Nous avons ainsi obtenu avec Bruno Salvy et Philippe Flajolet un cadre théorique à la fois général et puissant pour l'analyse en moyenne d'algorithmes [14]. À partir de spécifications telles que :

```
type Heap = leaf | min(key) Heap Heap;
     key = Latom(1);
     leaf = Latom(0);

procedure PathLength (h : Heap);
begin
   size(h);
   case h of
      (k,h1,h2) : begin PathLength(h1); PathLength(h2) end;
   end;
end;

procedure size (h : Heap);
begin
   case h of
      leaf : zero;
      (k,h1,h2) : begin one; size(h1); size(h2) end;
   end;
end;

measure one : 1;
        zero : 0;
```

cette méthode produit des « théorèmes automatiques » :

Théorème 2 *[14, Automatic Theorem 17] The average internal pathlength in a heap-ordered tree of size n is asymptotically*

$$2n \log n + (2\gamma - 3)n + 2\log n + \mathcal{O}(1).$$

Ma contribution à ce théorème est la transformation de la spécification ci-dessus en équations de séries génératrices, de dénombrement pour la structure de données *heap*, et de coût — les séries génératrices de coût sont aussi appelées *descripteurs de complexité* — pour la procédure PathLength ; ici ces séries (exponentielles) ont une forme explicite :

$$\mathrm{Heap}(z) = \frac{1}{1-z} \qquad \text{et} \qquad \tau\mathrm{PathLength} = -\frac{z}{z^2-2z+1} - \frac{2\log(1-z)}{z^2-2z+1}.$$

Grâce au programme EQUIVALENT de Bruno Salvy, un développement asymptotique du coefficient de z^n de ces séries est obtenu, d'où on déduit le coût asymptotique moyen du théorème 2.

J'ai proposé une première extension de cette méthode à l'analyse de fonctions à nombre fini de valeurs dans [25, ch. 4] et [26]. L'idée principale est que si les valeurs renvoyées par une fonction sont en nombre fini, on peut associer à chaque valeur une grammaire décrivant les entrées pour lesquelles la fonction donne cette valeur. (Ces différentes grammaires forment ainsi une partition de l'ensemble des données de la fonction.) Ceci permet de se ramener par simple réécriture à la classe des programmes sans fonction, que l'on sait analyser automatiquement. Cette extension est très puissante puisqu'elle autorise l'écriture de programmes utilisant des conditions `if f(a)=b then P(a) else Q(a)`, qui via la récursivité de la fonction f peuvent dépendre d'une partie non bornée de la structure a, alors qu'un simple filtrage ne teste qu'une partie bornée. Dans l'exemple ci-dessous, la fonction `pair` sélectionne les expressions arithmétiques ayant une valeur paire :

```
type N = zero | one | plus(N,N) | times(N,N);
     zero,one = atom(0);
     plus,times = atom(1);

function pair (n : N) : boolean;
begin
   case n of
      zero : true;
      one : false;
      plus(i,j) : if pair(i) then pair(j)
                  else if pair(j) then false else true;
      times(i,j) : if pair(i) then true else pair(j);
end;

procedure proba_pair (n : N);
begin
   if pair(n) then count
end;

measure count : 1;
```

À partir de ce programme, Lambda-Upsilon-Omega obtient le résultat suivant (seulement *semi*-automatique car MAPLE doit être assisté pour résoudre les équations produites).

Théorème 3 *[25, Théorème semi-automatique 10] La probabilité qu'une expression aléatoire de taille n tirée uniformément dans $\mathcal{N}$ soit paire est*

$$p_n = \frac{[z^n]\frac{2-\sqrt{2}\sqrt{1+\sqrt{1-16z}}}{4z}}{[z^n]\frac{1-\sqrt{1-16z}}{4z}},$$

et vérifie asymptotiquement

$$p_n = \frac{1}{\sqrt{2}} + \mathcal{O}(1/\sqrt{n}).$$

Pour cet exemple simple, on pourrait facilement produire à la main une grammaire des expressions arithmétiques paires et impaires :

```
type Neven = zero | plus(Neven,Neven) | plus(Nodd,Nodd)
           | times(Neven,N) | times(Nodd,Neven);
     Nodd = one | plus(Neven,Nodd) | plus(Nodd,Neven)
          | times(Nodd,Nodd);
```

Mais on s'aperçoit rapidement que cette grammaire peut être de taille exponentielle par rapport à la description sous forme de fonctions à nombre fini de valeurs, d'où l'intérêt de ces dernières pour rendre la description plus aisée. (La suite de l'analyse reste cependant inchangée, donc de complexité au pire exponentielle par rapport à la spécification.) Il serait d'ailleurs intéressant, en dehors du cadre strict de l'analyse d'algorithmes, d'avoir accès à ces grammaires-partitions produites en interne par Lambda-Upsilon-Omega.

Une seconde extension est proposée dans [29], pour la composition de fonctions. Là encore, la classe des programmes analysables automatiquement n'est pas étendue, seule la description de ces programmes est facilitée en se rapprochant des langages usuels. Illustrons cette extension sur un exemple. À partir de la spécification `expression` d'expressions arithmétiques,

```
type expression = zero | one | x | plus(expression,expression)
                | times(expression,expression);
```

et de la fonction `diff` de dérivation de telles expressions,

```
function diff (e : expression) : expression;
begin
   case e of
      plus(e1,e2) : plus(diff(e1),diff(e2));
      times(e1,e2) : plus(times(diff(e1),copy(e2)),
                          times(copy(e1),diff(e2)));
      zero : zero;
      one : zero;
      x : one;
   end;
end;
```

(la fonction `copy` recopiant une expression est définie de même) on peut spécifier la dérivation seconde simplement par :

```
function diff2 (e : expression) : expression;
begin
   diff(diff(e))
end;
```

Cette fonction est automatiquement traduite par Lambda-Upsilon-Omega en fonction équivalente sans composition (`plus` et `times` sont des opérateurs formant l'expression résultat, et non des fonctions) :

```
function diff_of_diff (e : expression) : expression;
begin
 case e of
  (plus,e1,e2):plus(diff_of_diff(e1),diff_of_diff(e2));
```

```
   (times,e1,e2):plus(plus(times(diff_of_diff(e1),copy_of_copy(e2)),
                           times(copy_of_diff(e1),diff_of_copy(e2))),
                      plus(times(diff_of_copy(e1),copy_of_diff(e2)),
                           times(copy_of_copy(e1),diff_of_diff(e2))));
   zero:zero;
   one:zero;
   x:zero;
  end;
 end;
```

où les trois fonctions `copy_of_diff`, `diff_of_copy`, `copy_of_copy` ont des descriptions analogues. On obtient donc 4 fonctions mutuellement récursives. Ce procédé a rendu possible (cf [29]) le développement asymptotique jusqu'à $k = 7$ de la taille moyenne $\tau\mathrm{diff}^{(k)}$ de la dérivée k^{e} d'une expression arithmétique, ce qui a permis de conjecturer la formule générale

$$\tau\mathrm{diff}_n^{(k)} = \frac{\Gamma(k/2+1)}{2^{k/2}} n^{k/2+1} + \mathcal{O}(n^{(k+1)/2}),$$

formule qui a ensuite été prouvée. Un autre résultat non trivial obtenu dans [29], toujours grâce à la composition des fonctions, est une expression régulière pour la représentation binaire des entiers se réduisant à 1 après exactement 3 transformations $n \to (3n+1)/2^k$, où 2^k est la plus grande puissance de 2 divisant $3n+1$:

$$\begin{aligned} S_3 \quad \to \quad & ((\epsilon|(100101111011010000)^* 1001011(\epsilon|1|1101|110110011|11011010000| \\ & 11011010000011))(100011)^* 1000|(100101111011010000)^* 100101 \\ & (\epsilon|11101100|1110110100000))(\epsilon|1)(10)^* 10^*. \end{aligned}$$

Ce calcul, relié évidemment à la conjecture de Collatz, rend effectif le résultat de Wilson et Shallit,[3] qui ont montré que les ensembles S_k sont 2-automatiques, donc s'expriment sous forme d'expressions régulières.

Perspectives. Les travaux sur l'analyse automatique d'algorithmes ont été poursuivis au sein du projet Algo de l'INRIA Rocquencourt avec le développement de la bibliothèque algolib.[4] L'utilisation d'un unique langage (en l'occurrence Maple) évite les problèmes d'interface rencontrés lors du développement de Lambda-Upsilon-Omega, les conventions d'écriture et de lecture de Maple changeant d'une version à une autre. La bibliothèque algolib regroupe en outre d'autres outils comme gfun (§2.1) et combstruct (§2.2).

Au niveau plus théorique, une des limitations principales de cette approche est que — malgré les extensions proposées — peu de programmes réels s'expriment sous la forme imposée par Lambda-Upsilon-Omega. Dans certains cas, un expert saura contourner cette difficulté : par exemple, une marche aléatoire qui ne descend jamais sous l'axe des abscisses (i.e. un mot de « grand Dyck » en termes combinatoires) se décompose en une suite d'arches suivie d'une demi-arche, ces arches s'exprimant elles-mêmes de façon récursive.[5]

---

[3] J. Shallit, D. Wilson, *The $3x+1$ problem and finite automata*, Bull. EATCS 46, 1992.

[4] `http://algo.inria.fr/libraries/software.html`

[5] J'ai pu utiliser cette expertise dans [20] pour l'analyse de la déviation moyenne par rapport à l'axe des abscisses d'une marche aléatoire dans le plan discret ; un des résultats de cet article est un algorithme

Mais pour nombre de programmes fonctionnant par effet de bord sur les données, comme par exemple le tri *shell-sort*, l'approche récursive semble ne pas pouvoir s'appliquer.

1.2 Génération aléatoire

Contexte. Des algorithmes efficaces de génération aléatoire sont connus depuis longtemps pour des structures de données particulières comme les mots de parenthèses.[6] Hickey et Cohen ont mis au point en 1983 deux algorithmes généraux[7] pour les grammaires *context-free*, l'un en espace linéaire et temps $\mathcal{O}(n^2 \log^2 n)$ pour produire une structure de taille n, l'autre en temps linéaire et espace mémoire $\mathcal{O}(n^{r+1})$ pour r non-terminaux. Mairson a amélioré en 1994 ce résultat avec un algorithme de complexité $\mathcal{O}(n^2)$ en temps et en espace.[8] D'autre part, Nijenhuis et Wilf ont montré dès 1978 comment générer de façon aléatoire et efficace des compositions ou partitions d'entiers.[9] Greene avait aussi observé dans sa thèse[10] que la description récursive de structures de données pouvait être utilisée pour générer des objets aléatoires.

Contribution. Reprenant et généralisant les idées de Wilf et Nijenhuis, nous avons mis au point avec Philippe Flajolet et Bernard Van Cutsem un algorithme efficace — en $\mathcal{O}(n \log n)$ opérations arithmétiques — pour la génération aléatoire uniforme de structures *décomposables* [15]. La méthode présentée s'applique aux structures étiquetées définies à partir d'objets atomiques par les constructeurs «union», «produit», «ensemble de» et «cycle de». Ces deux derniers constructeurs sont réduits sous forme *standard* grâce au constructeur de *marquage* Θ :

$$A = \mathrm{Set}(B) \Rightarrow \Theta A = A \cdot \Theta B, \quad A = \mathrm{Cycle}(B) \Rightarrow \Theta A = \mathrm{Sequence}(B) \cdot \Theta B.$$

Marquer un ensemble revient en effet à marquer un de ses éléments, le reste formant un nouvel ensemble ; de même, marquer un cycle (orienté) revient à marquer un de ses éléments, le reste formant une suite ordonnée. Une fois sous forme standard, les produits k-aires étant mis sous forme binaire via l'introduction de nouveaux non-terminaux — comme pour une forme normale de Chomsky —, on peut générer efficacement des objets aléatoires de façon uniforme simplement en suivant récursivement les probabilités associées à chaque branche de l'arbre correspondant à la construction de tels objets. Il suffit pour cela de déterminer au préalable le nombre a_k d'objets de chaque taille k dérivés par chaque non-terminal A. Chaque production de la forme standard se traduit directement en une

pour décomposer une série génératrice rationnelle bivariée en ses parties positive et négative par rapport à l'une des variables.

[6] D. B. Arnold, M. R. Sleep, *Uniform Random Generation of Balanced Parenthesis Strings*, ACM Transactions on Programming Languages and Systems, vol. 2, n° 1, 1980.

[7] T. Hickey, J. Cohen, *Uniform random generation of strings in a context-free language*, SIAM Journal of Computing vol. 12, n° 4, 1983.

[8] H. G. Mairson, *Generating Words in a Context Free Language Uniformly at Random*, Information Processing Letters, vol. 49, n° 2, 1994.

[9] A. Nijenhuis, H. S. Wilf, *Combinatorial Algorithms*, Academic Press, 2e édition, 1978.

[10] D. H. Greene, *Labelled formal languages and their uses*, PhD thesis, Stanford University, 1983, rapport STAN-CS-83-982.

procédure de génération, choisie parmi 6 modèles correspondant aux objets atomiques de taille 0 et 1, à l'union, au produit, au marquage et à son inverse. L'algorithme de génération aléatoire ainsi obtenu a une complexité arithmétique en $\mathcal{O}(n^2)$.

Pour un produit $C = A \cdot B$, la taille k de la première composante doit être choisie avec une probabilité proportionnelle au produit $a_k b_{n-k}$. La méthode *séquentielle* décrite ci-dessus parcourt les valeurs de k possibles dans l'ordre $0, 1, 2, ...$ Or l'espérance de k pour un tirage uniforme est en général linéaire en n, en particulier lorsque $A = B$. L'idée géniale consiste à parcourir les valeurs de k en suivant l'ordre *boustrophédon* $(0, n, 1, n-1, 2, n-2, ...)$. Le coût $f(n)$ de génération de structures de taille n vérifie alors la récurrence

$$f(n) \leq \max_k[f(k) + f(n-k) + \min(k, n-k)],$$

donnant un coût $\mathcal{O}(n \log n)$ non seulement en moyenne, mais surtout dans le pire cas, et de plus quelle que soit la structure considérée !

Théorème 4 *[15, Theorem 4.1] Any decomposable structure has a random generation routine that uses precomputed tables of size $\mathcal{O}(n)$ and achieves $\mathcal{O}(n \log n)$ worst-case time complexity.*

L'efficacité de cette méthode s'explique intuitivement de la façon suivante : un coût important pour le choix de k est contre-balancé par des tailles k et $n-k$ plus équilibrées lors des appels récursifs, ce qui donne au final une complexité arithmétique $\mathcal{O}(n \log n)$.

Une autre contribution de [15] est la mise en évidence d'un véritable *calcul symbolique* produisant un système d'équations pour les séries génératrices de coût, permettant ainsi de déterminer le coût moyen de la méthode séquentielle, et donc d'optimiser celle-ci, notamment de choisir le bon « sens » des produits $A \cdot B$, ou la décomposition associative optimale de $A \cdot B \cdot C$. Une idée connexe consiste à déséquilibrer artificiellement les produits $B \cdot B$ en leur appliquant l'opérateur de marquage Θ, la méthode séquentielle étant plus efficace pour $B \cdot \Theta B$ (méthode différentielle).

Nous avons ensuite étendu[11] avec Philippe Flajolet cette méthode, baptisée *méthode récursive*, aux structures non étiquetées. Tous ces résultats ont été implantés en MAPLE dans le logiciel GAÏA [28] devenu depuis COMBSTRUCT, et en MUPAD dans le logiciel CS. Cet aspect plus « logiciel » de mes travaux en génération aléatoire est décrit en §2.2.

Avec Laurent Alonso et Alain Denise, nous avons remarqué que l'un des facteurs bloquants de la génération aléatoire était la taille des nombres entiers a_n manipulés, généralement de taille n ou $n \log n$, et pouvant donc faire plusieurs centaines de chiffres. La génération d'un objet de taille n par la méthode boustrophédon décrite ci-dessus nécessite en effet un temps de calcul[12] $\mathcal{O}(n \log n M(n \log n))$, soit $\mathcal{O}(n^{3+o(1)})$ avec la multiplication naïve. Nous avons proposé une variante qui remplace les calculs entiers par des calculs

[11] Malheureusement, l'article décrivant le cas non étiqueté n'est pas terminé, mais plusieurs copies circulent « sous le manteau ».

[12] On considère à présent le temps de calcul (*bit complexity*), qui est le produit de la complexité arithmétique (*arithmetic complexity*) considérée jusqu'alors par le coût d'une opération sur les coefficients a_n. Si $a_n = \mathcal{O}(n \log n)$, ce coût est borné par $\mathcal{O}(M(n \log n))$ où $M(n)$ désigne le coût de la multiplication de nombres de n bits.

flottants (cette idée avait été évoquée par Mairson en 1994). Alain Denise et moi-même avons analysé précisément la probabilité d'erreur lorsqu'on utilise des nombres flottants :

Théorème 5 *[9, Theorem 4.3] The average bit-complexity of the ADZ method preprocessing, according to n and to the computer precision ε, is*

$$P_1(n,\varepsilon) = \mathcal{O}(n^2 M(\log \frac{1}{\varepsilon})) \ ;$$

the average bit-complexity for the generation of one structure is

$$C_1(n,\varepsilon) = \mathcal{O}\left(n \log n M(\log \frac{1}{\varepsilon}) + n^6 \varepsilon M(n \log n)\right),$$

where $M(x)$ stands for the cost of multiplying two x-bit numbers. [...]

En choisissant bien la précision ε, par exemple $\varepsilon = n^{-7}$, on obtient une complexité quasi-linéaire en n. Cette estimation est sans doute pessimiste, puisque $\varepsilon = n^{-3}$ semble suffisant en pratique : pour la génération d'arbres de Motzkin (unaires-binaires) de taille $n = 10^5$, l'utilisation de flottants machine (53 bits de mantisse) suffit dans plus de 99% des cas, les cas restants requérant davantage de précision. Cet algorithme a été implanté par Isabelle Dutour en MuPAD dans le logiciel CS [8] (§2.2).

Ce dernier travail illustre parfaitement mon intérêt pour les trois thèmes présentés dans ce mémoire, puisque partant d'un problème de génération aléatoire, nous avons utilisé les propriétés de l'arithmétique machine en double précision, et finalement implanté l'algorithme obtenu dans un logiciel de calcul formel.

Perspectives. Au niveau théorique, on ne sait pas traiter de manière aussi efficace des constructeurs tels que les ensembles sans répétition (dans le cas non étiqueté).[13] Une difficulté intrinsèque de ce constructeur est qu'il n'est pas véritablement « décomposable » : deux éléments d'un ensemble sans répétition ne peuvent pas être produits indépendamment, contrairement au cas des multi-ensembles. À mon avis, ce problème doit être abordé dans le cadre du « rang inverse » (voir ci-dessous). Le constructeur « substitution », très puissant, pose également problème, la méthode classique requérant des tables de $\mathcal{O}(n^2)$ coefficients. Au niveau logiciel, le problème principal, détaillé en §2.2, est de savoir comment mixer des algorithmes sous-quadratiques avec l'utilisation de flottants machine.

Sur un plan plus concret, la distribution uniforme de structures de données induite par une grammaire *context-free* ne correspond pas toujours à celle attendue par l'utilisateur. Ainsi le fait qu'un arbre binaire aléatoire ait une fois sur deux une branche réduite à un nœud surprend toujours. Il faudrait donc trouver des solutions pour pouvoir modéliser des distributions quelconques, ou restreindre certains paramètres des structures générées à certaines plages de valeurs.

Une autre piste, suivie par Gilles Schaeffer et Philippe Flajolet, consiste à utiliser un générateur aléatoire efficace, mais ne produisant pas des objets de taille fixée *a priori*. Supposons que l'on dispose d'un algorithme prenant en entrée un paramètre entier m

[13] J'ai cependant donné dans [30] une méthode permettant de générer un arbre à sous-arbres distincts — correspondant à la spécification `A = Prod(Z, PowerSet(A))` — de taille 200 en quelques dizaines de secondes. Mais ce résultat est obtenu au prix d'un précalcul en $\mathcal{O}(n^3)$, ce qui est rédhibitoire en pratique.

et produisant en sortie des structures aléatoires de taille X_m, où X_m est une variable aléatoire. Si on sait analyser précisément la distribution de X_m, on peut en déduire la valeur optimale de m pour obtenir en sortie une taille n donnée. Si X_m est suffisamment « concentrée », cela fournit un algorithme efficace. Cette approche a été employée avec succès par Gilles Schaeffer pour certaines classes de graphes.[14] Cette méthode fait partie des algorithmes « avec rejet », le critère de rejet étant ici la taille de la structure produite. Dans d'autres cas, on rejette les objets n'ayant pas les bonnes propriétés, par exemple un ensemble ayant deux éléments identiques si l'on désire un ensemble sans répétition. Là encore, pour obtenir un algorithme efficace, il est crucial que les « bonnes » structures aient une densité suffisamment grande, et pour déterminer cette densité, des méthodes puissantes d'analyse d'algorithmes sont essentielles.

Rang inverse. Une question étroitement liée à la génération aléatoire est le problème du *rang inverse*. Étant donnée une classe $\mathcal{A}$ d'objets combinatoires, telle que pour tout entier $n \geq 0$, le sous-ensemble $\mathcal{A}_n$ des objets de taille n est fini — soit alors A_n son cardinal —, une fonction de rang inverse est une fonction injective :

$$f : (n, 1 \leq k \leq A_n) \to a \in \mathcal{A}_n.$$

Comme card$\mathcal{A}_n = A_n$, c'est en fait une bijection. Le terme « rang inverse » provient du fait que la fonction inverse f^{-1} caractérise chaque objet de $\mathcal{A}_n$ par un unique entier $k \in [1, A_n]$, appelé son « rang ». Un algorithme de rang inverse est une méthode effective implantant une telle fonction f. Trouver un tel algorithme est intéressant dans la mesure où cela fournit immédiatement une procédure de génération aléatoire uniforme pour la classe $\mathcal{A}$ à partir d'un générateur aléatoire uniforme d'entiers. Il suffit en effet de donner comme argument à f la taille n désirée, et un entier aléatoire dans l'intervalle $[1, A_n]$.

Nous avons proposé avec François Bertault une méthode générale pour le rang inverse de structures décomposables non étiquetées [3]. C'est un premier résultat général dans le domaine des structures non étiquetées, qui peut sans doute être amélioré. L'idée de base, décrite par François Bertault dans sa thèse de doctorat,[15] consiste à utiliser comme structure intermédiaire la notion de « forme ». Une forme est une structure de données indiquant la taille de l'objet à générer, le nombre de ses composantes ainsi que la taille de chacune de celles-ci. Le rang inverse de structures décomposables se ramène ainsi au problème de rang inverse sur les formes ; ce dernier, selon la structure des objets, se ramène au rang inverse sur des listes ou ensembles, pour lesquels des algorithmes classiques sont connus.[16] Nous obtenons aussi via les formes un algorithme de rang inverse *incrémental* de coût amorti constant :

Théorème 6 *[3, Theorem 5.5] For specifications using context-free grammars, i.e. using only* Union *and* Prod *constructors of arity* 2*, the amortized cost of the* `next_unranking` *function is* $\mathcal{O}(1)$ *if there are* $\Omega(n^2)$ *structures of size* n.

[14] G. Schaeffer, *Conjugaison d'arbres et cartes combinatoires aléatoires*, thèse de doctorat, Université Bordeaux I, 1998.

[15] F. Bertault, *Génération et tracé de structures décomposables*, thèse de l'université Henri Poincaré Nancy 1, 1997.

[16] A. Nijenhuis, H. S. Wilf, *Combinatorial Algorithms*, Academic Press, 2e édition, 1978.

Une implantation en MAPLE de cet algorithme nous a permis de générer automatiquement les 24 ensembles de colliers bicolores de taille 6, chaque collier ayant au moins 3 perles.

1.3 Détection de fautes d'orthographe : EPELLE

CONTEXTE. La commande SPELL du système d'exploitation Unix permet la détection de fautes d'orthographe en anglais. Elle est basée sur un algorithme utilisant une table de racines usuelles, accompagnée de tables de préfixes et suffixes.[17] À la fin des années 1980, aucun logiciel équivalent n'existait sous Unix pour le français. Ma motivation principale pour en développer un provenait du cours d'algorithmique et de structures de données de Philippe Flajolet.

CONTRIBUTION. Le principe du logiciel EPELLE [27] diffère de celui de SPELL. Au lieu de considérer plusieurs tables (racines, préfixes et suffixes), EPELLE prend en entrée *une seule* table, contenant toutes les formes correctes possibles dans la langue considérée. De ce fait, ce programme peut être utilisé directement pour n'importe quelle langue — sans avoir à définir préfixes ni suffixes courants — ou même pour n'importe quel ensemble de mots. L'inconvénient majeur est que la liste exhaustive des mots corrects peut s'avérer très longue. On pallie ce problème en utilisant une représentation sous forme d'arbre dictionnaire (*dictionary trie*), avec un partage maximal des suffixes. Cette représentation est construite efficacement par une méthode d'insertion incrémentale via une fonction de hachage (voir [27] pour les détails techniques). Ainsi, une table interne de 506Ko suffit à stocker les quelques 260000 formes françaises de la version 2.3 d'EPELLE, soit près de 3Mo sous forme ASCII :

```
% wc -l dico
 260689 dico
% time ./zpellin < dico > dico.z
260689 words, 563526 nodes, 84269 nodes after compaction
1.440u 0.100s 0:01.57 98.0%     0+0k 0+0io 77pf+0w
% ls -l dico*
-rw-r--r--   1 zimmerma      2952153 Jun  6 21:19 dico
-rw-r--r--   1 zimmerma       505614 Jun  6 21:20 dico.z
```

PERSPECTIVES. La représentation interne sous forme de *trie* d'EPELLE utilise des listes pour stocker les différents choix possibles à un endroit de l'arbre. Il existe une structure de données théoriquement plus efficace, à savoir les *tries* hybridés par des arbres binaires de recherche, ou *ternary search tries*. Ainsi le coût de recherche pour un nœud ayant n fils est $\mathcal{O}(\log n)$ au lieu de $\mathcal{O}(n)$. Julien Clément a modifié EPELLE afin d'utiliser des *ternary search tries* à la place des listes.[18] La nouvelle version obtenue gagne effectivement en efficacité, mais aussi en place mémoire utilisée par la table interne.

D'autres optimisations restent sans doute possibles, mais sur des exemples réels, on se rend vite compte que la phase de vérification des mots dans le dictionnaire est peu coûteuse

[17] J. A. Bentley, *A spelling checker*, Communications of the ACM, vol. 28, n° 5, 1985.

[18] J. Clément, *Arbres digitaux et sources dynamiques*, thèse de doctorat, Université de Caen, septembre 2000.

par rapport aux autres phases (transformation des accents d'un système de codage à un autre, suppression des commentaires ou commandes spécifiques au langage utilisé comme par exemple LaTeX, tri des mots restants).

Ce travail a montré que des structures de données évoluées comme le *trie* ou sa variante *ternary search trie* sont très utiles — elles sont employées dans d'autres domaines comme la recherche de motifs en génomique — dès lors que l'on sait les manipuler efficacement.

1.4 Estimations asymptotiques

Suite à une question posée par Bernard Mourrain, nous avons étudié avec Isabelle Dutour et Laurent Habsieger le nombre de chemins nord-est dans le plan discret, de pente bornée et de largeur fixée [11]. Le problème posé par Bernard Mourrain est lié au calcul des solutions d'un système d'équations polynomiales et à la borne de Bezout. Plus précisément, soient $d \in \mathbb{N} \setminus \{0,1\}$, $\mathcal{F}_{d,n}$ le sous-ensemble des $(a_1, \ldots, a_n) \in \mathbb{N}^n$ défini par

$$\left\{ \begin{array}{ccc} a_1 & \leq & d-1 \\ a_1 + a_2 & \leq & 2(d-1) \\ \vdots & & \vdots \\ a_1 + \cdots + a_n & \leq & n(d-1) \end{array} \right.$$

et soit $g_{n,k}$ le nombre d'éléments de $\mathcal{F}_{d,n}$ tels que $a_1 + \cdots + a_n = k$. Si l'on considère que chaque a_j correspond à un déplacement vers le nord de a_j pas et vers l'est d'un pas, chaque $(a_1, \ldots, a_n)$ correspond à un chemin de n pas vers l'est et de $a_1 + \cdots + a_n$ pas vers le nord, restant toujours en dessous de la droite de pente $d-1$. La question posée par Bernard Mourrain, liée au rang de la matrice Bezoutien de n polynômes, consiste à obtenir une estimation asymptotique en n et d de

$$G_{d,n} = \sum_{0 \leq k \leq n(d-1)} \min(g_{n,k}, g_{n,n(d-1)-k}) \, .$$

Une majoration grossière donne $G_{d,n} = \mathcal{O}(e^n d^n)$. Nous avons obtenu une estimation plus précise, par exemple pour d impair :

Théorème 7 *[11] Soit d impair, et $c_d = (\frac{d+1}{2})^{(d+1)/2} (\frac{2}{d-1})^{(d-1)/2}$, alors*

$$\sqrt{\frac{\lfloor \frac{d+1}{2} \rfloor}{2\pi n^3 \lfloor \frac{d-1}{2} \rfloor}} \, c_d^{n+1} \, (1 + o(1)) \leq G_{d,n} \leq \sqrt{\frac{2}{\pi n}} \, \frac{d+1}{d-1} \, c_d^n \, (1 + o(1)).$$

Ce résultat pourrait avoir un lien avec un vieil article de Motzkin communiqué par Mireille Bousquet-Mélou.[19]

Mes recherches en algorithmique, présentées dans ce chapitre, ont porté principalement sur l'analyse en moyenne et la génération aléatoire. Si mes recherches actuelles, décrites dans les chapitres suivants, se situent plus en calcul formel et en arithmétique, ma formation initiale en algorithmique reste extrêmement utile.

[19] A. Dvoretzky, Th. Motzkin, *A Problem of Arrangements*, Duke Math. J. 14, 1947.

Chapitre 2

Calcul formel

Mes activités en calcul formel ont eu principalement pour but de mettre à la disposition de la communauté scientifique des outils de manipulation d'objets symboliques, soit correspondant à mes recherches, soit en implantant de nouveaux algorithmes afin de mieux les comprendre. Ces implantations ont été faites dans les logiciels de calcul formel MAPLE et MuPAD, et plus récemment dans les outils plus spécialisés NTL de Victor Shoup et RS de Fabrice Rouillier. Le livre [17] écrit avec Claude Gomez et Bruno Salvy montre comment utiliser le logiciel MAPLE pour traiter des problèmes issus de divers domaines des mathématiques. J'ai également implanté lors de mon séjour à Paderborn en 1994-1995 plusieurs algorithmes fondamentaux de calcul symbolique, ce qui m'a notamment permis d'apprendre ces algorithmes et donc de savoir « ce qu'il y a dans la boîte » [16]. J'ai aussi contribué à l'évaluation des logiciels existants dans des domaines précis [21].

Ce chapitre commence par décrire en §2.1 la réalisation du programme GFUN de manipulation de fonctions holonomes, puis en §2.2 le volet logiciel de mes travaux en génération aléatoire (cf §1.2), avec les outils COMBSTRUCT et CS. Les deux sections suivantes portent sur deux problèmes majeurs pour la manipulation de polynômes univariés : la factorisation en §2.3, et l'isolation des racines réelles en §2.4. Enfin, les travaux de synthèse et de vulgarisation sont mentionnés en §2.5.

2.1 Fonctions holonomes : GFUN

Ma première implantation de grande envergure en calcul formel fut le logiciel GFUN de manipulation de fonctions et récurrences holonomes, écrit avec Bruno Salvy [24]. GFUN contient d'une part des fonctions permettant, à partir d'une liste de coefficients, de « deviner » une série génératrice, sous forme explicite (fraction rationnelle ou hypergéométrique) ou implicite (équation algébrique, différentielle ou récurrence), étendant ainsi la fonction `convert/ratpoly` de MAPLE :

```
> with(share): with(gfun):
> deq := listtodiffeq([3, 19, 193, 2721, 49171, 1084483, 28245729,
  848456353, 28875761731], y(z));
                              2             2 /d       \
deq := [{-3 - z + (1 - 6 z - z ) y(z) - 4 z  |-- y(z)|, y(0) = 3}, ogf]
                                             \dz      /
```

D'autre part GFUN contient des fonctions de manipulation de fonctions *holonomes* (solutions d'équations différentielles à coefficients polynomiaux) :

```
> rec := diffeqtorec(deq[1], y(z), u(n));
     {u(0) = 3, -u(n) + (-10 - 4 n) u(n + 1) + u(n + 2), u(1) = 19}
```

ainsi qu'un « compilateur » produisant des fonctions efficaces de calcul des coefficients de ces séries :

```
> foo := rectoproc(rec, u(n));
foo := proc(n)
local i, u0, u1, u2;
   u0 := 3;
   u1 := 19;
   for i from 2 to n-1 do u2:= u0 + 2*u1 + 4*u1*i; u0:= u1; u1:= u2
   od;
   u0 + 2*u1 + 4*u1*n
end
> seq(foo(i), i=0..11);
3, 19, 193, 2721, 49171, 1084483, 28245729, 848456353,

    28875761731, 1098127402131, 46150226651233, 2124008553358849
```

Nous nous sommes contentés d'implanter dans GFUN des algorithmes classiques de la littérature. Ce logiciel s'est pourtant révélé extrêmement utile, à la fois pour nos propres recherches, mais aussi pour la communauté scientifique. Il continue à être utilisé puisque des « bugs » sont encore signalés sur la liste `gfun@inria.fr`. Le principal utilisateur est d'ailleurs Neil Sloane, qui utilise GFUN pour son programme *SuperSeeker*,[20] essayant de multiples transformations pour ramener une suite d'entiers donnée par l'utilisateur à une suite connue.

Au delà de l'intérêt du logiciel en lui-même, la leçon que j'ai tirée de GFUN est qu'en calcul formel, il ne faut pas chercher à tout prix des formes closes : des formules implicites s'avèrent tout aussi intéressantes, dès lors que l'on dispose d'outils efficaces de manipulation de celles-ci.

2.2 Génération aléatoire : COMBSTRUCT et CS

Isabelle Dutour a implanté dans le système de calcul formel MUPAD l'algorithme de génération aléatoire utilisant des nombres flottants développé avec Alain Denise (§1.2) ; le logiciel correspondant s'appelle CS [8]. Comme son équivalent COMBSTRUCT en MAPLE, à partir d'une spécification de structure de données, CS permet de compter le nombre d'objets de taille donnée,

```
>> Schroeder := [S, {S=Union(Z, Prod(Z, Sequence(S, card>=2)))}]:
>> cs::count(Schroeder, size=100);

                  63071286725499290051879587077009179781061461
```

[20] `http://www.research.att.com/~njas/sequences/`

ou bien de générer un objet aléatoire de taille donnée :

```
>> cs::draw(Schroeder, size=17);

Prod(Z, Sequence(Z, Z, Prod(Z, Sequence(Z, Prod(Z, Sequence(Z, Z)),

   Z)), Z, Prod(Z, Sequence(Prod(Z, Sequence(Z, Z, Z, Z)), Z))))
```

Par rapport à COMBSTRUCT, la particularité de CS est la possibilité de produire directement un programme en langage C utilisant les flottants-machine pour la génération aléatoire efficace de telles structures :

```
>> cs::compile(Schroeder[2], target=C, file="schroeder.c", main=S);
```

Le programme `schroeder.c`, une fois compilé, permet la génération rapide de structures de données de grande taille, pouvant aller jusqu'à un million en moins d'une minute pour les arbres de Schröder ci-dessus.

CS est donc un véritable compilateur de MuPAD vers C. Le thème de la compilation de sous-langages du calcul formel vers des langages de plus bas niveau comme C ou FORTRAN est du plus grand intérêt. Il est illusoire de vouloir compiler n'importe quel programme de calcul formel, mais certaines classes bien définies donnent lieu à une compilation efficace, comme nous l'avons montré dans le cas de la génération aléatoire. Un autre exemple est la génération de code C ou FORTRAN pour des calculs matriciels via les programmes MACROC ou MACROFORT, développés en MAPLE par Patrick Capolsini et Claude Gomez, et portés tous deux en MuPAD par Tony Scott.

PERSPECTIVES. En ce qui concerne la complexité arithmétique, si la génération aléatoire d'un objet est quasi-linéaire, l'étape limitante reste le précalcul des tables de coefficients, qui est quadratique. Joris van der Hoeven a montré comment effectuer ce précalcul avec une complexité arithmétique quasi-linéaire via la multiplication *détendue* de séries formelles.[21] Isabelle Dutour a implanté l'algorithme de van der Hoeven en CS, mais uniquement dans l'interprète, qui manipule des entiers en précision arbitraire. Il n'est pas possible de s'en servir tel quel dans le compilateur, car les algorithmes de multiplication rapide dégradent fortement la qualité numérique des approximations flottantes. On a donc actuellement le choix entre l'utilisation de CS entièrement sous MuPAD, de complexité $\mathcal{O}(M(n)^2)$ pour le précalcul, et la génération de code C, de complexité $\mathcal{O}(n^2)$. Il faudrait donc comprendre comment combiner efficacement les algorithmes rapides sur les séries formelles avec des coefficients flottants, sans trop dégrader la qualité des résultats numériques.

2.3 Factorisation de polynômes

CONTEXTE. La factorisation de polynômes de $\mathbb{Z}[x]$ comporte quatre étapes : (i) élimination des facteurs carrés via pgcd ; (ii) factorisation modulo un entier premier p ; (iii) remontée de cette factorisation modulo p^k (*Hensel lifting*) ; (iv) recombinaison des facteurs modulaires pour trouver les vrais facteurs sur $\mathbb{Z}[x]$. Pour la dernière étape, deux variantes

[21] J. van der Hoeven, *Relax, but don't be too lazy*, `http://www.math.u-psud.fr/~vdhoeven`, 1999.

existent : l'algorithme de Zassenhaus, de complexité exponentielle dans le cas le pire, et un algorithme polynomial utilisant la réduction des réseaux inventé par Lenstra, Lenstra et Lovácz.[22] La plupart des implantations utilisent toujours l'algorithme de Zassenhaus, car l'algorithme de Lenstra, Lenstra et Lovácz, malgré une meilleure complexité asymptotique, reste plus coûteux sur les polynômes que l'on parvient *actuellement* à factoriser. Ceci reste vrai même dans le pire cas de l'algorithme de Zassenhaus, à savoir pour les polynômes de Swinnerton-Dyer,

$$\prod(x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \cdots \pm \sqrt{p_j}),$$

où p_j désigne le j^{e} nombre premier. Ces polynômes, de degré 2^j, se décomposent en produits de facteurs de degré au plus 2 modulo tout nombre premier, mais sont irréductibles dans $\mathbb{Z}[x]$.

Contribution. En 1999, avec Victor Shoup et John Abbott, nous avons amélioré l'algorithme de recombinaison de Zassenhaus. S'il y a r facteurs modulaires, l'algorithme de Zassenhaus teste 2^r combinaisons dans le pire cas, c'est-à-dire lorsque le polynôme est irréductible. Notre amélioration, présentée à ISSAC'2000 [1], réduit la complexité de cette étape à $\mathcal{O}(r2^{r/2})$. Les deux ingrédients essentiels sont d'une part l'utilisation de tables (*memory stacks*) pour stocker les valeurs correspondant à certaines combinaisons de facteurs, d'autre part l'utilisation astucieuse d'opérations sur les entiers machine pour effectuer de l'arithmétique d'intervalles en virgule fixe.

Si $a_1, ..., a_r$ sont les coefficients de degré $d-1$ des facteurs modulo p^k — on désigne de façon générique par d le degré, non nécessairement identique, de ces polynômes —, la combinaison des l facteurs modulaires d'indices $i_1, i_2, ..., i_l$ a comme coefficient de degré $d-1$:

$$a_{i_1} + a_{i_2} + \cdots + a_{i_l} \bmod p^k.$$

Lorsque cette combinaison donne un vrai facteur, l'entier obtenu après réduction modulo p^k est petit,[23] alors qu'une combinaison ne donnant pas un facteur de $\mathbb{Z}[x]$ produit un coefficient pouvant prendre n'importe quelle valeur entre $-p^k/2$ et $p^k/2$. L'idée est donc de tabuler des sommes partielles $a_{i_1} + a_{i_2} + \cdots + a_{i_m} \bmod p^k$ afin d'accélérer la recherche d'éventuelles bonnes combinaisons. Si on tabule ces sommes jusqu'à $i_m \le s$, la complexité passe de 2^r à $s2^{r-s}$ au prix d'un espace mémoire $\mathcal{O}(2^s)$.[24] Cette méthode s'étend facilement au test des coefficients de degré $d-2$ (voir [1]).

L'amélioration que nous avons proposée reste une heuristique, car dans le cas où la somme des coefficients de degré $d-1$ d'une combinaison des $r-s$ derniers facteurs modulaires « correspond » à une valeur dans la table, il faut *a priori* parcourir les 2^s combinaisons des s premiers facteurs pour identifier la partition correspondante. Cependant en pratique le gain est très significatif, même sur les polynômes de Swinnerton-Dyer, comme le montre l'implantation faite par Victor Shoup dans la bibliothèque NTL.[25] Le

[22] A. K. Lenstra, H. W. Lenstra, L. Lovácz, *Factoring Polynomials with Rational Coefficients*, Mathematische Annalen, vol. 261, 1982.

[23] On effectue ici une réduction symétrique, c'est-à-dire entre $-p^k/2$ et $p^k/2$.

[24] Le parcours des $r-s$ facteurs restants coûte 2^{r-s}, et la recherche dichotomique dans la table coûte s.

[25] `http://www.shoup.net/ntl/`

principal succès de cette implantation a été obtenu sur un polynôme de degré 972 communiqué par Jean-Charles Faugère.[26] La factorisation de ce polynôme, ayant au moins 54 facteurs modulaires modulo tout « petit » nombre premier p, est hors d'atteinte de l'algorithme de Zassenhaus, 2^{54} étant de l'ordre de $1.8 \cdot 10^{16}$. Avec $s = 24$, soit une table de quelques centaines de Mo, l'implantation en NTL de notre méthode prouve que ce polynôme est irréductible en une heure environ sur une machine actuelle.

Perspectives. En juillet 2000, Mark van Hoeij a annoncé une nouvelle méthode pour la recombinaison des facteurs modulaires, qui a créé une petite révolution dans le domaine de la factorisation de polynômes.[27] L'algorithme de van Hoeij est basé sur la réduction de réseau comme celui de Lenstra, Lenstra et Lovácz, la différence essentielle étant que le réseau considéré est de taille égale au nombre r de facteurs modulaires, et non au degré n du polynôme à factoriser. Par exemple, pour le polynôme de Jean-Charles Faugère mentionné plus haut, la taille du réseau à réduire passe de 972 à 54 ! Sachant que dans le pire cas $r \leq n/2$, et que la complexité de l'algorithme LLL de réduction des réseaux est en $\mathcal{O}(n^{5+\varepsilon})$, on comprend tout l'intérêt de cette variante.[28]

L'algorithme de van Hoeij étend en quelque sorte l'amélioration proposée avec Abbott et Shoup. Au lieu de considérer uniquement les coefficients de degré $d-1$ ou $d-2$, van Hoeij considère toutes les traces des recombinaisons, traces qui s'obtiennent linéairement à partir des facteurs modulaires. Il étend conjointement les calculs aux nombres p-adiques : ainsi un « vrai » facteur s'identifie par des traces qui sont des entiers. L'utilisation astucieuse de la réduction des réseaux permet de grouper progressivement les facteurs modulaires nécessairement associés dans une « vraie » recombinaison, et ainsi d'obtenir en quelques étapes de réduction de réseau la factorisation sur $\mathbb{Z}[x]$. Avec Guillaume Hanrot, nous avons implanté en NTL l'algorithme de van Hoeij. Avec cette implantation, le polynôme de degré 972 de Jean-Charles Faugère, impossible à factoriser il y a deux ans, et pour lequel il fallait de l'ordre d'une heure via l'amélioration proposée avec Abbott et Shoup, est maintenant factorisé en quelques secondes seulement !

À ma connaissance, le caractère polynomial de l'algorithme de van Hoeij n'a pas encore été prouvé. Cependant, cet algorithme fonctionne si bien qu'on peut considérer le problème de la factorisation de polynômes sur $\mathbb{Z}[x]$ comme clos en pratique. Il reste maintenant à identifier les classes de polynômes (en une ou plusieurs variables, avec des coefficients entiers ou sur une extension algébrique) pour lesquelles l'algorithme de van Hoeij s'applique.

2.4 Isolation des racines réelles d'un polynôme

Contexte. Isoler les racines réelles d'un polynôme univarié à coefficients entiers est un problème fondamental du calcul formel. Cela permet par exemple de compter le nombre

[26] J.-C. Faugère, *How my computer find all the solutions of Cyclic 9*, Rapport LIP6 007, 2000.

[27] M. van Hoeij, *Factoring polynomials and the knapsack problem*, Journal of Number Theory, à paraître, `http://www.math.fsu.edu/~hoeij/papers.html`.

[28] Plus précisément, le thèorème 16.11 de *Modern Computer Algebra* donne pour la réduction d'un réseau de taille n avec des coefficients bornés en valeur absolue par A un coût de $\mathcal{O}(n^4 \log A)$ opérations sur des entiers de taille $\mathcal{O}(n \log A)$, soit une complexité en temps de $\mathcal{O}(n^4 \log A \cdot M(n \log A)) = \mathcal{O}(n^{5+\varepsilon} \log^{2+\varepsilon} A)$.

total de racines réelles, ou dans un intervalle donné, ou encore d'obtenir par dichotomie une approximation numérique très précise de ces racines. Deux classes de méthodes exactes existent : celles utilisant les suites de Sturm — basées sur l'opérateur de dérivation $P \to \frac{\partial P}{\partial x}$ — et celles utilisant la règle de Descartes — basées sur l'opérateur de translation $P \to P(x+1)$. Ces deux méthodes sont implantées en MAPLE :

```
> p := -85*x^5-55*x^4-37*x^3-35*x^2+97*x+50:
> readlib(sturm)(sturmseq(p, x), x, -infinity, infinity);
                                        3
> readlib(realroot)(p);
                         [[0, 2], [-1, 0], [-2, -1]]
```

Le résultat renvoyé par `sturm` indique que p admet 3 racines sur $\mathbb{R}$, alors que celui renvoyé par `realroot` — utilisant la règle de Descartes — isole ces trois racines via les intervalles $[-2,-1]$, $[-1,0]$ et $[0,2]$.

La règle de Descartes exprime que le nombre de racines positives d'un polynôme est borné par le nombre de changements de signe dans la suite de ses coefficients. Le polynôme ci-dessus a un seul changement de signe dans la suite $-85, -55, -37, -35, 97, 50$ de ses coefficients, donc au plus une racine positive. Vincent en 1836 a énoncé une sorte de réciproque de la règle de Descartes, et on attribue à Uspensky en 1948 le premier véritable algorithme basé sur cette règle.[29] Depuis, plusieurs auteurs, notamment Collins et Akritas, Johnson et Krandick, ont proposé des variantes et améliorations de l'« algorithme d'Uspensky ».

CONTRIBUTION. Dans l'article [23], nous donnons avec Fabrice Rouillier une présentation unifiée — sous forme de parcours d'arbre — des méthodes basées sur la règle de Descartes, notamment l'algorithme de Collins et Akritas, et celui de Krandick. En remarquant que les polynômes $P(\frac{x+c}{2^k})$ nécessaires à l'algorithme peuvent être calculés directement à partir du polynôme initial $P(x)$ et de manière efficace, nous déduisons une version optimale quant à l'occupation mémoire : cette version ne stocke en effet qu'un nombre constant de polynômes, alors que la variante de Collins et Akritas (respectivement de Krandick) stocke un nombre de polynômes proportionnel à la profondeur (respectivement à la largeur) de l'arbre de recherche associé à la méthode d'Uspensky. Nous donnons aussi une version optimale à la fois en mémoire et en temps de calcul, et calculant exactement la même suite de polynômes que la version de Collins et Akritas. La remarque cruciale est le fait que dans un parcours en « profondeur d'abord » (*depth first search*) d'un arbre binaire, deux nœuds consécutifs sont reliés par une relation très simple, par conséquent une homothétie et une translation unitaire suffisent pour passer d'un polynôme à l'autre :

Proposition 1 *Dans l'algorithme*

`GenericDescartes(initializeTreeConst`, `getNewNodeRel`, `addSuccessorsConst`, $<_{\text{back}}$)

toutes les translations dans la procédure `getNewNodeRel` *sont soit* T_0*, soit* T_1*. En d'autres termes, si* (k,c) *et* (k',c') *sont deux nœuds consécutifs dans la liste ordonnée* $[(k_i,c_i)]_{i=0..s}$

[29] J. V. Uspensky, *Theory of Equations*, McGraw-Hill 1948. Cette attribution est contestée par Akritas dans son article *There is no "Uspensky's method"*, actes de SYMSAC'86, Waterloo, Ontario, ACM, 1986.

— par rapport à l'ordre $<_{\text{back}}$ *— représentant* `Tree`(P)*, alors l'expression* $2^{k-k'}c' - c$ *vaut soit* 0*, soit* 1.

Si la complexité en temps reste inchangée, le gain en occupation mémoire est très sensible, et permet donc de résoudre des problèmes inaccessibles jusqu'alors. Nous avons pu ainsi isoler avec moins d'1 Mo de mémoire les racines du polynôme de Wilkinson de degré 500 et du polynôme de Mignotte de degré 200, alors que l'algorithme de Collins et Akritas nécessite au moins 36 Mo pour les mêmes calculs, et celui de Krandick au moins 50 Mo.

Perspectives. Le parcours d'arbre effectué par l'algorithme d'Uspensky est, sauf polynômes pathologiques, quasi-optimal. De plus, plusieurs astuces évitent les branches inutiles. Les améliorations potentielles concernent donc les calculs effectués pour passer d'un nœud au suivant. Il s'agit principalement d'une translation $P(x) \to P(x+1)$, coûtant $\sim n^2/2$ additions de coefficients, où n est le degré de P. Des algorithmes sous-quadratiques existent[30] pour de telles translations, mais ne semblent pas utilisables ici, car ils nécessitent des multiplications de coefficients, plus coûteuses au total que les additions de la translation classique.

Une autre idée est de diminuer le coût des translations en ne considérant que les bits de poids fort des coefficients, par exemple via une arithmétique d'intervalles à précision variable, et en augmentant cette précision lorsqu'elle est insuffisante pour décider du signe d'un coefficient lors de l'application de la règle de Descartes. Fabrice Rouillier développe une telle version « hybride ».

Une dernière piste consiste à généraliser la règle de Descartes à une autre base de polynômes, qui permettrait des translations plus rapides. Par exemple, le calcul de $P(x+1)$ dans la base des factorielles décroissantes $x(x-1)...(x-n+1)$ s'effectue en $\mathcal{O}(n)$ additions via la « table des différences », au lieu de $\mathcal{O}(n^2)$ dans la base des x^n. Ainsi, un analogue de la règle de Descartes dans la base des factorielles décroissantes pourrait diminuer d'un facteur n la complexité totale. On pourrait aussi utiliser les polynômes de Bernstein $B_k^n(x) = \binom{n}{k}x^k(1-x)^{n-k}$, vérifiant $B_k^n(x+1) = B_{n-k}^n(-x)$.

2.5 Travaux de « vulgarisation »

Mon principal travail de vulgarisation scientifique fut la rédaction en 1994 et 1995, avec Claude Gomez et Bruno Salvy, du livre « Calcul formel : mode d'emploi. Exemples en Maple » [17]. À l'inverse de la plupart des ouvrages sur le calcul formel, nous avions choisi dès le départ de partir des domaines d'applications, et non des commandes du système. C'est sans doute la raison pour laquelle cet ouvrage reste d'actualité, même si la syntaxe de Maple a changé depuis.

Outre ce livre, j'ai rédigé avec Frank Postel le chapitre « *Solving Ordinary Differential Equations* » [21] de l'ouvrage collectif « *Computer Algebra Systems : A Practical Guide* » édité par Michael Wester. Ce chapitre, comparant les solveurs d'équations différentielles ordinaires d'Axiom, Derive, Maple, Mathematica, Macsyma, MuPAD et

[30] J. von zur Gathen, J. Gerhard, *Fast algorithms for Taylor shifts and certain difference equations*, ISSAC'97, pages 40–47.

Reduce, indique quels types d'équations ou de systèmes (linéaires ou non, équations admettant une solution explicite, ou se récrivant en un autre type d'équation) les différents logiciels savent résoudre, ou simplement manipuler.

En 1997, j'ai été sollicité par Nicole Berline et Claude Sabbah pour faire un exposé sur le calcul formel aux enseignants de classes préparatoires aux grandes écoles lors des journées « X-UPS ». L'article correspondant intitulé « Calcul formel : ce qu'il y a dans la boîte » [31] décrit sommairement cinq algorithmes de calcul symbolique : les algorithmes de sommation de Gosper et de Zeilberger, les algorithmes de Berlekamp et de Zassenhaus[31] de factorisation dans $\mathbb{F}_p[x]$ et $\mathbb{Z}[x]$, enfin l'algorithme ECM de factorisation d'entiers dû à Lenstra. Le but de ce court article était de faire comprendre aux utilisateurs les méthodes utilisées par les systèmes de calcul formel. En effet, l'un des aspects déconcertants par rapport au calcul numérique est le fait que changer un signe + en − peut faire basculer d'une classe de problèmes à une autre. Pour l'intégration de fonctions algébriques, l'exemple ci-dessous, dû à Manuel Bronstein, illustre bien cette particularité : la fonction $\frac{t+1}{(t-2)\sqrt{t^3+1}}$ admet une primitive élémentaire, ce qui n'est pas le cas pour $\frac{t-1}{(t-2)\sqrt{t^3+1}}$.

```
G82268 (3) -> integrate((t+1)/(t-2)/sqrt(t^3+1), t)

                     +------+
                     | 3          3      2
               (6t + 6)\|t  + 1  + t  + 12t  - 6t + 10
          log(---------------------------------------)
                         3     2
                        t  - 6t  + 12t - 8
   (3)  - --------------------------------------------
                                3

G82268 (4) -> integrate((t-1)/(t-2)/sqrt(t^3+1), t)

            t
           ++        %Q - 1
   (4)     |   ------------------ d%Q
          ++            +-------+
                        |  3
              (%Q - 2)\|%Q  + 1
```

Pour bien comprendre cette difficulté, il est indispensable d'appréhender les algorithmes utilisés en calcul formel et les classes de problèmes auxquelles ils s'appliquent. Je déplorais ainsi dans [31] l'absence d'ouvrage décrivant les principaux algorithmes du calcul formel ; ce manque a été comblé avec l'excellent livre « *Modern Computer Algebra* » de von zur Gathen et Gerhard, et plusieurs autres ouvrages plus spécialisés (celui de Manuel Bronstein sur l'intégration symbolique de fonctions transcendantes, celui de Petkovšek, Wilf et Zeilberger sur la sommation symbolique, ...)

La présentation chronologique de mes travaux en calcul formel montre une double orientation : d'une part de problèmes plus exotiques (fonctions holonomes, génération

[31] Cet algorithme est maintenant dépassé (cf §2.3).

aléatoire) vers des problèmes plus fondamentaux du calcul formel (polynômes univariés), d'autre part de travaux d'implantation d'algorithmes classiques vers la mise au point de nouveaux algorithmes. Cette orientation va de concert avec le passage de logiciels généraux, comme MAPLE ou MuPAD, à des outils plus spécialisés comme NTL ou RS, écrits en C ou C++. Cette tendance à m'intéresser à des aspects de plus en plus fondamentaux du calcul formel, nécessitant par conséquent des outils de plus bas niveau pour une implantation efficace, se poursuit dans mes travaux en arithmétique, décrits dans le chapitre suivant.

Chapitre 3

Arithmétique

Mon attrait pour l'arithmétique — en particulier en précision arbitraire — ne date pas d'hier, puisqu'en 1987 je réalisais ma première implantation des algorithmes de multiplication entière de Karatsuba et de Schönhage, sous la direction de Jean Vuillemin. Mon intérêt plus récent pour l'arithmétique flottante « exacte », dont la signification sera précisée plus loin, provient de mon expérience des logiciels de calcul formel, en tant qu'utilisateur. J'avais en effet remarqué que ces logiciels ne sont pas tout-à-fait satisfaisants dès que l'on effectue des calculs flottants. D'une part le résultat d'un même calcul peut varier d'un logiciel à l'autre, dépendant en particulier de la base interne du système (2 pour la majorité des systèmes, et 10 pour MAPLE). D'autre part le même logiciel donne même parfois des résultats différents sur des machines différentes. Enfin il est rarement — voire jamais — possible de déduire une information *certaine* d'un calcul flottant, même en augmentant la précision de calcul. Ces trois défauts ont une seule et même origine : le manque de spécification mathématique précise des calculs flottants faits par ces logiciels.

Une telle spécification, le standard[32] IEEE 754, existe pour les calculs en double précision (dans les langages C ou Fortran par exemple) ; ceci a révolutionné les calculs numériques depuis une quinzaine d'années. La norme IEEE 754 exige l'*arrondi exact* — on dit aussi arrondi correct — pour les quatre opérations de base et la racine carrée, c'est-à-dire que le résultat d'un calcul doit être le nombre machine le plus proche de la valeur exacte, suivant le mode d'arrondi donné par l'utilisateur. Il est donc naturel de souhaiter — Keith Geddes, l'un des développeurs de MAPLE, m'a aussi exprimé ce souhait — un logiciel de calcul formel dont les calculs flottants soient spécifiés rigoureusement.[33] En particulier il serait très agréable de pouvoir indiquer avec quel mode d'arrondi doit être effectuée chaque opération flottante, comme c'est le cas dans la norme IEEE 754. Du rêve à la réalité, il n'y a qu'un pas !

Deux faits ont servi de catalyseur pour — essayer de — franchir ce pas. D'une part

[32] *IEEE Standard for Binary Floating-Point Arithmetic ANSI-IEEE Standard 754-1985*, 1985, New York, approuvé le 21 mars 1985 par le *IEEE Standards Board*, et le 26 juillet 1985 par le *American National Standards Institute*.

[33] MAPLE 6 autorise des arrondis dirigés, mais l'implantation laisse encore à désirer (`Rounding :=0` positionne l'arrondi vers zéro) :

```
> Digits :=3 : a :=1.0 : b :=9e-5 : Rounding :=0 : a-b ;
                         1.0
```

les travaux avec Alain Denise sur la génération aléatoire utilisant des calculs flottants (§1.2), et le développement du logiciel CS avec Alain Denise et Isabelle Dutour (§2.2), d'autre part l'action de recherche coopérative « Outils pour un calcul numérique fiable », et notamment la coopération avec l'équipe SAAO (aujourd'hui Arénaire) de Jean-Michel Muller. Le développement de CS a démontré l'intérêt des flottants pour remplacer les entiers, ainsi que l'efficacité des calculs en nombres machine.

Cependant, nous avions buté sur une limitation importante des flottants double-précision, qui ne peuvent pas représenter des nombres plus grands que $2^{1024} \sim 1.8 \cdot 10^{308}$. Ceci pose problème pour les applications en calcul symbolique où les entiers de plusieurs centaines voire milliers de chiffres sont chose courante. Il a donc fallu pour CS fabriquer des flottants avec une plus large plage d'exposants. Nous avons choisi de représenter l'exposant par un mot signé de 32 bits, la mantisse étant représentée par un flottant machine. Cela repousse la limite à des nombres d'environ 646 millions de chiffres décimaux. Cela devrait être suffisant pour la plupart des applications... L'implantation correspondante, du nom de DPE pour *Double Plus Exponent*, peut bien sûr être utilisée indépendamment de CS ; elle a d'ailleurs été intégrée par Fabrice Rouillier dans le logiciel RS. DPE permet de représenter des intervalles de nombres flottants et d'effectuer les opérations arithmétiques de base sur ces intervalles. Initialement prévue uniquement pour des nombres positifs, cette bibliothèque a été étendue en 2000 à des nombres quelconques, avec une base paramétrable. Par exemple, le nombre d'arbres binaires de taille $n = 1000$ — soit le nombre de CATALAN $\frac{1}{n}\binom{2n-2}{n-1}$ — est inclus dans l'intervalle en base 2 :

```
1.1696822755901528*2^1982..1.1696822755948506*2^1982
```

ou en base 10 :

```
5.1229405377312798*10^596..5.1229405377539834*10^596
```

L'article de synthèse [35] décortique en une trentaine de pages les principaux algorithmes connus pour effectuer des calculs entiers ou flottants, qu'ils soient naïfs ou de meilleure complexité asymptotique, mais aussi les algorithmes intermédiaires comme celui de Karatsuba pour la multiplication, qui ont souvent une large plage d'utilisation. Pour ce qui est des calculs flottants, cet article se limite aux opérations de base. L'un de mes objectifs est, parallèlement au développement de la bibliothèque MPFR, de détailler[34] les algorithmes qui y sont implantés pour le calcul des fonctions élémentaires et spéciales. Un tel document compléterait pour la précision arbitraire l'excellente bibliographie de Lozier et Olver[35] et le projet DLMF[36] de version électronique du fameux *Handbook of Mathematical Functions* d'Abramowitz et Stegun.

Ce chapitre est organisé comme suit : en §3.1 sont décrits les travaux sur la division et la racine carrée entières, primitives de base pour les mêmes opérations sur les nombres flottants (§3.2), et qui ont été intégrées à la bibliothèque GNU MP (GMP).[37] Ces recherches fondamentales ont été motivées par le développement de la bibliothèque MPFR

[34] Y compris l'analyse d'erreur garantissant l'arrondi exact.

[35] D. W. Lozier, F. W. J. Olver, *Numerical Evaluation of Special Functions*, National Institute of Standards and Technology, rapport NISTIR 5383, 1994.

[36] `http://dlmf.nist.gov/`

[37] `http://www.swox.se/gmp/`

d'arithmétique flottante en précision arbitraire (§3.3). Enfin un premier résultat reliant l'arithmétique des ordinateurs à la théorie des nombres est décrit en §3.4.

3.1 Division et racine carrée entières

Contexte. Dans un rapport de recherche[38] datant de 1998, Christoph Burnikel et Joachim Ziegler présentent un algorithme de division récursive sur les entiers. Cet algorithme — qui prolonge des travaux de Moenck et Borodin,[39] puis Jebelean[40] — fournit quotient et reste de la division d'un nombre de $2n$ chiffres par un nombre de n chiffres, en essentiellement deux fois le temps d'une multiplication $n \times n$ via l'algorithme de Karatsuba. Après avoir implanté cet algorithme dans la bibliothèque GNU MP, j'ai naturellement cherché un algorithme similaire pour la racine carrée, autre primitive de base de la couche `mpn` de GNU MP.

Contribution. Le rapport [33] décrit un algorithme de racine carrée récursive sur les entiers : étant donné $n \geq 0$, il renvoie $s, r \geq 0$ tels que $n = s^2 + r$ avec $0 \leq r \leq 2s$, ce qui correspond à l'arrondi vers zéro :

Algorithm `SqrtRem`$(n = a_3 b^3 + a_2 b^2 + a_1 b + a_0)$
Input : $0 \leq a_i < b$ with $a_3 \geq b/4$
Output : (s, r) such that $s^2 \leq n = s^2 + r < (s+1)^2$
  $(s', r') \leftarrow$ `SqrtRem`$(a_3 b + a_2)$
  $(q, u) \leftarrow$ `DivRem`$(r'b + a_1, 2s')$
  $s \leftarrow s'b + q$
  $r \leftarrow ub + a_0 - q^2$
  if $r < 0$ **then**
    $r \leftarrow r + 2s - 1$
    $s \leftarrow s - 1$
  return (s, r)

L'implantation[41] en GNU MP a permis de gagner un facteur deux par rapport à la version précédente de `mpn_sqrtrem`, basée quant à elle sur l'itération de Newton. La note [34] donne une preuve « sur papier » des implantations que j'ai faites en GNU MP de cet algorithme de racine carrée, ainsi que de l'algorithme de division rapide de Moenck-Borodin-Jebelean-Burnikel-Ziegler. Avec l'aide d'Yves Bertot, Loïc Pottier, Laurence Rideau, et de l'assistant de preuve Coq,[42] nous avons obtenu une preuve « sur machine » de la validité de la description mathématique de cet algorithme.

Perspectives. Prouver la spécification d'un algorithme ne garantit rien quant à son implantation ; prouver cette dernière est bien plus difficile car il faut notamment vérifier

[38] Ch. Burnikel, J. Ziegler, *Fast Recursive Division*, rapport de recherche MPI-I-98-1-022, Saarbrücken, 1998.

[39] R. Moenck, A. Borodin, *Fast modular transforms via division*, 13th Annual IEEE Symposium on Switching and Automata Theory, 1972.

[40] T. Jebelean, *Practical Integer Division with Karatsuba Complexity*, ISSAC'97, 1997.

[41] Je suis un peu fier de celle-ci, notamment de la routine principale `mpn_dq_sqrtrem` qui implante cet algorithme en place.

[42] `http://pauillac.inria.fr/coq/`

la gestion correcte de la mémoire, ce qui est non trivial pour les grands entiers découpés en plusieurs mots-machine. C'est le travail de titan auquel Yves Bertot, Didier Bondyfalat et Nicolas Magaud se sont attaqués : la preuve des implantations en GNU MP de la division et de la racine carrée récursive, y compris la gestion de la mémoire.

À partir d'un algorithme de racine carrée entière, comme montré dans [35], on obtient immédiatement la racine carrée flottante. Étant donnés s et r tels que $n = s^2 + r$ avec $0 \le r \le 2s$, il est facile de déterminer l'arrondi exact de $\sqrt{n}$. Pour l'arrondi vers 0 ou vers $-\infty$, c'est s ; pour l'arrondi vers $+\infty$, c'est s si $r = 0$ ou $s+1$ sinon ; enfin pour l'arrondi au plus proche, c'est s si $r \le s$ et $s+1$ sinon (le carré de $s+1/2$ n'étant pas entier, le problème de l'arrondi pair ne se pose pas).

La racine carrée récursive n'est en fait qu'une simple itération de Newton, avec calcul incrémental du terme d'erreur. Un des problèmes pour une utilisation en pratique pour la double précision est la nécessité de disposer d'une fonction de division entière, relativement lente sur les processeurs actuels. Or l'itération classique pour $1/\sqrt{n}$ n'utilise que des multiplications. On peut donc se demander s'il existe une version analogue — avec calcul exact et incrémental du terme d'erreur — pour $1/\sqrt{n}$.

3.2 Division et racine carrée flottantes

Contexte. Soit l'itération de Newton pour le calcul de l'inverse de a :

$$x_{k+1} = x_k + x_k(1 - ax_k).$$

Si x_k est une approximation de $1/a$ avec n chiffres corrects, alors pour obtenir une approximation x_{k+1} avec $2n$ chiffres, il faut prendre les $2n$ chiffres de poids fort de a, que l'on multiplie avec les n chiffres de x_k, ce qui donne un produit ax_k de $3n$ chiffres. On sait d'avance que les n chiffres de poids fort de ce produit donnent 1 ; d'autre part les n chiffres de poids faible n'influent pas sur x_{k+1}. Il est donc naturel de se demander si l'on peut calculer *directement* les n chiffres du milieu. Par exemple, si $a = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + \mathcal{O}(t^4)$ et $x = x_0 + x_1 t + \mathcal{O}(t^2)$, alors

$$ax = a_0x_0 + (a_0x_1 + a_1x_0)t + (a_1x_1 + a_2x_0)t^2 + (a_2x_1 + a_3x_0)t^3 + \mathcal{O}(t^4),$$

et on voudrait par exemple calculer directement les termes en t^2 et t^3. Il est clair que c'est possible en n^2 opérations, mais est-il possible de le faire plus rapidement avec un algorithme à la Karatsuba ?

Contribution. La réponse est « oui ». Dans l'exemple ci-dessus, on calcule $\alpha = (a_1 + a_2)x_1$, $\beta = a_2(x_0 - x_1)$, $\gamma = (a_2 + a_3)x_0$, et les coefficients de t^2 et t^3 du produit ax sont donnés respectivement par $\alpha + \beta$ et $\gamma - \beta$, soit via 3 multiplications au lieu de 4 :

```
> alpha:= (a1+a2) * x1: beta:= a2 * (x0-x1): gamma:= (a2+a3) * x0:
> expand([alpha + beta, gamma - beta]);
                          [x1 a1 + a2 x0, x0 a3 + x1 a2]
```

Ceci se généralise à des tailles quelconques ; l'algorithme MP (pour *Middle Product*), mis au point avec Guillaume Hanrot et Michel Quercia [18], calcule directement les n termes intermédiaires d'un produit de $2n$ termes par n termes :

Algorithm `MP`$([a_0, ..., a_{2n-1}], [x_0, ..., x_{n-1}])$
0. If $n = 1$, return $[a_0 x_0]$
1. $\alpha =$ `MP`$([a_0 + a_{n/2}, ..., a_{n-1} + a_{3n/2-1}], [x_{n/2}, ..., x_{n-1}])$
2. $\beta =$ `MP`$([a_{n/2}, ..., a_{3n/2-1}], [x_0 - x_{n/2}, ..., x_{n/2-1} - x_{n-1}])$
3. $\gamma =$ `MP`$([a_{n/2} + a_n, ..., a_{3n/2-1} + a_{2n-1}], [x_0, ..., x_{n/2-1}])$
4. $h = \alpha + \beta$
5. $l = \gamma - \beta$
6. Return the concatenation of h and l

Théorème 8 *[18, Th. 1] Algorithm* `MP` *returns* $[c_{n-1}, ..., c_{2n-2}]$*, where* $c_k = \sum_{i+j=k} a_i x_j$*, using exactly* $K(n) = n^{\log_2 3}$ *ring multiplications.*

Cet algorithme peut être utilisé pour accélérer les itérations de Newton comme ci-dessus, mais aussi directement comme remarqué par Michel Quercia pour la division et la racine carrée flottantes.

PERSPECTIVES. Michel Quercia a montré récemment que $\mathrm{MP}(n) = M(n)$, c'est-à-dire que si l'on dispose d'un algorithme multipliant deux polynômes de n termes sur un anneau commutatif en $M(n)$ opérations, alors on peut transposer cet algorithme en une méthode de calcul du produit médian de polynômes de $2n-1$ et n termes, et vice-versa. Ce résultat non seulement résout le problème de l'analyse de complexité de MP, mais aussi étend sa portée aux algorithmes asymptotiquement optimaux, puisque l'identité $\mathrm{MP}(n) = M(n)$ vaut aussi pour la FFT.

L'algorithme MP, initialement mis au point dans un cadre arithmétique, devrait trouver sa place plus généralement en calcul formel, puisqu'il s'applique aussi bien aux polynômes qu'aux séries formelles. Cela montre à nouveau le lien étroit existant entre mes trois thèmes de recherche.

3.3 La bibliothèque MPFR

CONTEXTE. Depuis le programme MP écrit en Fortran par Richard Brent à la fin des années 1970,[43] qui donc précédait l'apparition de la norme IEEE 754, personne n'avait essayé de donner une sémantique précise au calcul en précision arbitraire. Les logiciels de calcul en précision arbitraire, à la fois ceux de calcul formel (Maple, Mathematica, ...) et les bibliothèques spécialisées (Pari, CLN, ...) n'offrent pas de sémantique précise aux calculs flottants, mais se contentent de garantir « plus ou moins » une erreur d'au plus une unité sur le dernier chiffre significatif. En termes plus techniques, il s'agit d'une erreur d'au plus un *ulp* (*unit in last place*).

Un réel besoin existait donc. D'ailleurs d'autres groupes de chercheurs y ont répondu parallèlement à nous : la bibliothèque Arithmos[44] fournit l'arrondi exact avec une base

[43] R. P. Brent, *A Fortran Multiple-Precision Arithmetic Package*, Transactions on Mathematical Software 4, 1978.

[44] `http://win-www.uia.ac.be/u/cant/arithmos/`, université d'Anvers, Belgique.

interne pouvant varier de 2 à 2^{24}, mais n'implante actuellement aucune fonction transcendante. La version 6 de MAPLE propose plusieurs modes d'arrondi en base 10, mais l'implantation (ou la spécification) laisse à désirer comme on l'a vu page 25.

CONTRIBUTION. Après de fructueuses discussions, notamment avec Jean-Michel Muller, Joris van der Hoeven et Torbjörn Granlund, nous[45] nous sommes lancés début 1999 dans la réalisation de MPFR [7], une bibliothèque de calcul flottant en précision arbitraire permettant à l'utilisateur de choisir au bit près la précision de chaque variable, ainsi que le mode d'arrondi de chaque opération (vers zéro, vers moins l'infini, vers plus l'infini, ou au plus proche). Pour des raisons d'efficacité et de portabilité, MPFR est construite au-dessus de la couche `mpn` de GNU MP. La version actuelle (MPFR 2001) implante les quatre opérations arithmétiques de base (addition, soustraction, multiplication, division), la racine carrée, l'exponentielle et le logarithme, le sinus et le cosinus, la moyenne arithmético-géométrique, les conversions et entrées-sorties (changements de base), les constantes $\log 2$ et π. De nombreuses nouvelles fonctions sont déjà disponibles dans la version de développement : l'exponentielle en base 2, l'arc-tangente, les sinus, cosinus et tangente hyperboliques et leurs inverses, et la factorielle. Notre objectif est de fournir fin 2001 une version implantant toutes les fonctions mathématiques du standard C9X.

PERSPECTIVES. MPFR est pour l'instant surtout un prototype montrant la faisabilité d'une arithmétique flottante en précision arbitraire avec une sémantique formelle, sans perte notable d'efficacité. La comparaison avec des outils spécialisés pour une précision donnée, comme la bibliothèque QD[46] (*Quad Double*) de Bailey *et al* — qui ne garantit pas l'arrondi exact — n'est pas au désavantage de MPFR. Les applications d'une telle bibliothèque sont multiples : arithmétique d'intervalles, calcul certifié de constantes mathématiques, émulation d'arithmétiques en précision fixe. Nathalie Revol et Fabrice Rouillier ont déjà réalisé à partir de MPFR une bibliothèque MPFI d'arithmétique d'intervalles, elle-même utilisée par Nathalie Revol pour déterminer le maximum global d'une fonction sur un domaine fini.

3.4 Liens avec la théorie des nombres

CONTEXTE. De nombreux problèmes d'arrondi exact, une fois traduits en termes arithmétiques, se ramènent à des problèmes de théorie des nombres, dont certains sont non-triviaux. En particulier, la recherche de pires cas pour l'arrondi exact correspond à l'identification de points du plan discret proches de courbes, qui peuvent être algébriques ou transcendantes. Trouver des bornes inférieures fines pour ce genre de problème, ou mieux mettre au point des méthodes trouvant les points les plus proches de telles courbes, autorise la mise au point d'algorithmes efficaces pour l'arrondi correct de fonctions algébriques ou transcendantes.

[45] La « MPFR *team* », initialement composée de Guillaume Hanrot, Fabrice Rouillier et moi-même, s'est élargie depuis avec les contributions de Sylvie Boldo, David Daney, Mathieu Dutour, Emmanuel Jeandel, Vincent Lefèvre et Nathalie Revol.

[46] `http://www.nersc.gov/~dhbailey/mpdist/qd/qd.html`

Un exemple de tel problème d'arithmétique des ordinateurs est l'existence d'un inverse pour l'arrondi au plus proche : étant donné un nombre flottant x avec une mantisse de k bits, existe-t-il y de précision k également tel que xy, une fois arrondi au plus proche sur k bits, donne exactement 1. Jean-Michel Muller s'est intéressé à ce problème[47] et a montré qu'il se ramenait à la question suivante : étant donné $2^k \leq m < 2^k$, existe-t-il $2^k \leq n < 2^k$ tel que $2^{2k-1} - 2^{k-2} \leq mn \leq 2^{2k-1} + 2^{k-1}$? Des expérimentations l'ont amené à conjecturer une densité limite de $\frac{1}{2} - \frac{3}{2}\log\frac{4}{3}$ pour les nombres sans inverse.

Contribution. Avec Gérald Tenenbaum, Joël Rivat et Guillaume Hanrot, nous avons prouvé dans [19] cette conjecture :

Théorème 9 *Pour* $r = 0, 1, 2$, *soit* $\gamma_r(k)$ *le nombre de* $x \in F_k \cap [1, 2]$ *ayant exactement* r *FP-inverse(s). Alors*

$$\begin{aligned}
\gamma_0(k)/2^{k-1} &= \tfrac{1}{2} - \tfrac{3}{2}\log\tfrac{4}{3} + \mathcal{O}(2^{-k/3}) = 0.0684768917... + \mathcal{O}(2^{-k/3}) \\
\gamma_1(k)/2^{k-1} &= 1 - \tfrac{3}{2}\log\tfrac{9}{8} + \mathcal{O}(2^{-k/3}) = 0.8233254464... + \mathcal{O}(2^{-k/3}) \\
\gamma_2(k)/2^{k-1} &= -\tfrac{1}{2} + \tfrac{3}{2}\log\tfrac{3}{2} + \mathcal{O}(2^{-k/3}) = 0.1081976622... + \mathcal{O}(2^{-k/3}).
\end{aligned}$$

Nous avons également obtenu un résultat similaire pour la racine carrée inverse : la densité des nombres de $[1, 2[$ (respectivement $[\frac{1}{2}, 1[$) admettant une racine carrée inverse tend vers $\frac{3\sqrt{2}-3}{2}$ (respectivement $\frac{3\sqrt{2}-3}{2\sqrt{2}}$). La preuve fait intervenir des résultats classiques de théorie des nombres (expression en termes de la fonction de Bernoulli B_1, réduction à un problème de sommes d'exponentielles via un résultat de Vaaler, et utilisation d'un résultat de van der Corput pour borner cette somme).

Perspectives. Certaines questions de théorie des nombres issues de problèmes apparemment simples d'arithmétique des ordinateurs semblent hors de portée des techniques actuelles. Une telle question est la suivante : étant donné un nombre flottant x avec une mantisse de n bits tel que $y = \frac{1}{\sqrt{x}}$ n'est pas représentable exactement sur n bits, quel est le nombre maximum de 0 — ou de 1 — consécutifs dans le développement binaire de y, après les n premiers bits ? Si b_n est ce nombre, on sait juste que b_n est borné par $2n + o(1)$.[48] En termes arithmétiques, cela revient à chercher des bornes inférieures pour $|2^{3n-1} - x^2 y|$ où x et y sont des entiers de n bits. Un tel exemple pour $n = 21$ est $x = 1721436$, $y = 1556245$, qui donnent :

$$x^2 y = 2^{62} + 1616.$$

Cette identité donne lieu à $1/\sqrt{y}$ dont le développement binaire contient 31 bits à « 1 » après les 21 premiers bits :

```
> Digits:=61: convert(evalf(1/sqrt(1556245)), binary);
                                                                -10
.1101001000100010110111111111111111111111111111111111000000000 10
```

[47] J.-M. Muller, *Some algebraic properties of floating-point arithmetic*, actes de RNC'4, Dagstuhl, avril 2000, `http://www.dagstuhl.de/DATA/Events/00/00162.proceedings/papers/p16.ps`

[48] T. Lang, J.-M. Muller, *Bounds on Runs of Zeros and Ones for Algebraic Functions*, actes d'ARITH'15, Vail, Colorado, IEEE Computer Society, 2001.

Chapitre 4

Calculs de l'extrême

Les « calculs de l'extrême » m'ont toujours captivé, moins comme une fin en soi que comme un moyen de se confronter aux meilleurs chercheurs actuels, et parfois d'inventer de nouveaux algorithmes. En effet, pour battre un record, il ne suffit pas de disposer d'une machine très rapide, ni d'implanter le meilleur algorithme connu, il faut combiner les deux. Mon attrait porte donc à la fois sur la partie algorithmique et sur l'implantation efficace.

Les problèmes décrits ici tournent autour de la combinatoire énumérative (problème des n reines en §4.1), de l'arithmétique (sommes de cubes en §4.2 et suites de nombres premiers en §4.3), de la cryptographie (factorisation d'entiers en §4.4), et des générateurs pseudo-aléatoires (trinômes irréductibles en §4.5).

D'autres calculs ont été omis soit parce qu'ils n'ont pas été couronnés de succès (par exemple l'extension des 5 suites aliquotes commençant par un entier inférieur à 1000 et de statut inconnu) soit parce que ma contribution s'est limitée au développement de logiciel (par exemple le programme `gmp-ecm` [32] de factorisation d'entiers par la méthode des courbes elliptiques, qui détient le record actuel pour cette méthode avec un facteur de 54 chiffres trouvé par Nik Lygeros et Michel Mizony).

4.1 Les n reines

Le problème des n reines — comment placer n reines ne s'attaquant pas sur un échiquier $n \times n$? — m'a toujours fasciné. Dès 1987, lors de mes débuts en algorithmique avec le langage Pascal, au cours d'une séance de travaux dirigés sous la direction de Philippe Flajolet — qui devait par la suite devenir mon directeur de thèse — j'ai découvert une méthode de génération aléatoire de solutions au problème des n reines pour de grands échiquiers (taille 1000 voire 10000). L'idée, qui a sans doute déjà été énoncée par ailleurs, est la suivante : au lieu de parcourir l'arbre de recherche de la gauche vers la droite, on parcourt les branches partant de chaque nœud interne dans un ordre aléatoire. Dans le cas des n reines, un nœud interne correspond au placement de reines sur les colonnes 1 à j, et ses fils aux lignes autorisées pour le placement de la reine en colonne $j+1$; une solution facile à implanter consiste à choisir de façon aléatoire l'indice i de la première ligne essayée, puis à parcourir les autres en ordre circulaire : $i+1, ...n, 1, 2, ..., i-1$. Cette méthode qui amortit les « effets de bord » — au sens propre ici — fonctionne bien lorsque la longueur de cheminement dans l'arbre de recherche n'est pas trop grande par rapport au nombre de solutions. Cet exercice fut précurseur de mes recherches en génération aléatoire (§1.2).

L'un de mes premiers « calculs de l'extrême » fut celui de $T(23) = 128850048$, le nombre de solutions du problème des n reines sur un échiquier toroïdal pour $n = 23$ [22]. Ilan Vardi m'avait « initié » au problème des n reines sur tel échiquier,[49] qui n'a de solution que pour n premier avec 6. Un tel calcul s'étalant sur plusieurs mois est toujours périlleux, surtout lorsque la répartition des calculs et la compilation des résultats sont en partie manuelles. J'ai appris depuis que le résultat de ce calcul avait été confirmé.[50] Le record actuel est $T(25) = 1957725000$, et pour un échiquier classique $Q(23) = 24233937684440$. Ces valeurs confortent la conjecture énoncée dans [22] :

Conjecture 1

$$\lim_{\substack{n\to\infty \\ (n,6)=1}} \frac{T(n)}{n \log n} = \alpha > 0, \qquad \lim_{n\to} \frac{Q(n)}{n \log n} = \beta > 0.$$

Il est un peu frustrant de ne pas disposer d'algorithme plus efficace que la recherche exhaustive — modulo quelques améliorations par ci par là — pour calculer $Q(n)$ ou $T(n)$. Rivin et Zabih ont pourtant proposé une méthode originale,[51] basée sur le calcul des termes sans carré du permanent d'une matrice bien choisie, dont le temps de calcul est borné par 8^n. Si on croit à la conjecture ci-dessus, cet algorithme est plus rapide que la recherche exhaustive. Malheureusement l'espace mémoire nécessaire (en 4^n) rend cet algorithme inutilisable en pratique.

4.2 Sommes de cubes

Il s'agit là d'un travail effectué en collaboration avec François Bertault et Olivier Ramaré [2]. Nous avons montré que tout entier compris entre 1290741 et $3.375 \cdot 10^{12}$ peut s'écrire comme somme d'au plus 5 cubes. Ce résultat, obtenu par recherche exhaustive grâce au soutien du Centre Charles Hermite, a été amélioré depuis par Jean-Marc Deshouillers, François Hennecart et Bernard Landreau, qui ont montré que tout entier de l'intervalle $[1290741, 10^{16}]$ est somme de 5 cubes. La méthode utilisée par Deshouillers, Hennecart et Landreau consiste dans un premier temps à déterminer des plages effectives où tous les entiers congrus à $k \bmod 9$ sont somme de 4 cubes — par exemple tout $n \in [4 \cdot 10^8, 4.5 \cdot 10^{11}]$ congru à $0 \bmod 9$ est somme de 4 cubes — puis à étendre ces résultats pour les sommes de 5 cubes grâce au lemme suivant.

[49] Contrairement à l'échiquier classique ayant $2n - 1$ diagonales montantes (respectivement descendantes), l'échiquier toroïdal n'a que n diagonales montantes (respectivement descendantes).

[50] Voir la référence $A051906$ de l'encyclopédie électronique des suites d'entiers, `http://www.research.att.com/~njas/sequences/`

[51] I. Rivin, R. Zabih, *A dynamic programming solutions to the N-queens problems*, Information Processing Letters 41, 1992.

Lemme 1 (Deshouillers, Hennecart, Landreau, 1996) *Si tout* $n \in [a, b]$, $(b-a \geq 9)$, *congru à* $i \bmod 9$ *est somme de* 4 *cubes alors*

1. *tout* $n \in [a, a + \frac{(b-a)^{3/2}}{27}]$ *congru à* $i \bmod 9$ *est somme de* 5 *cubes,*
2. *tout* $n \in [a, a + \frac{(b-a)^{3/2}}{27}]$ *congru à* $i + 1 \bmod 9$ *est somme de* 5 *cubes,*
3. *tout* $n \in [a, a + \frac{(b-a)^{3/2}}{27} - \frac{(b-a)}{3}]$ *congru à* $i - 1 \bmod 9$ *est somme de* 5 *cubes.*

Ce lemme réduit la complexité de la vérification que tous les entiers dans l'intervalle $[1290741, N]$ sont somme de 5 cubes de $\mathcal{O}(N)$ — complexité de la méthode exhaustive que nous avions utilisée — à $\mathcal{O}(N^{2/3})$. Il est naturel de songer à se ramener derechef aux sommes de 3 cubes, ce qui permettrait d'avoir un algorithme en $\mathcal{O}(N^{4/9})$; malheureusement cela échoue car les intervalles ne contenant que des sommes de 3 cubes sont de taille trop réduite, même en se restreignant à des classes modulaires.

Ce problème reste d'ailleurs non complètement résolu. Deshouillers, Hennecart et Landreau ont formulé la conjecture suivante :

Conjecture 2 (Deshouillers, Hennecart, Landreau, 1996) [52] *Tout entier supérieur à 7 373 170 279 850 est somme de 4 cubes.*

Cette conjecture, et par suite le nombre ci-dessus qui est le plus grand connu n'étant pas somme de 4 cubes, a été obtenue en considérant les différentes classes modulo 63, et en cherchant pour chaque classe la plus grande exception n suivie d'un intervalle $[n+1, 10n]$ sans exception. La preuve de cette conjecture semble actuellement hors de portée.

Une autre question non résolue concerne la densité des sommes de 3 cubes : est-elle nulle ou strictement positive ? Deshouillers, Hennecart et Landreau ont observé des phénomènes curieux : la densité expérimentale — avec sommants distincts — qui vaut 0.099902 jusqu'à $2 \cdot 10^{12}$, accuse un minimum vers $3 \cdot 10^{12}$ avant d'augmenter sensiblement.[53] Leur modèle fonctionne en revanche très bien pour les sommes de 4 bicarrés.[54]

4.3 Premiers consécutifs en progression arithmétique

Ce problème, énoncé par Richard Guy dans le chapitre A6 de son livre *Unsolved Problems in Number Theory* — et posé par un lecteur du magazine *Pour la Science* en 1997 — consiste à trouver k nombres premiers consécutifs en progression arithmétique. Soient $p_1, p_2 = p_1 + a, p_3 = p_1 + 2a, ..., p_k = p_1 + (k-1)a$ ces k entiers. Il est facile de voir que la raison a de la progression est forcément un multiple de tous les nombres premiers p inférieurs ou égaux à k, dès lors que $p_1 > k$. En effet, si a est premier avec p, c'est un générateur de $\mathbb{Z}/p\mathbb{Z}$, et l'un des p_j est forcément multiple de p. Avec Harvey Dubner, Tony Forbes, Nik Lygeros et Michel Mizony [10], nous avons trouvé des progressions de huit et neuf nombres, et une de dix nombres de 93 chiffres :

[52] J.-M. Deshouillers, F. Hennecart, B. Landreau, I G. P. Purnaba, 7 373 170 279 850, Mathematics of Computation, vol. 69, n° 229, 2000.

[53] J.-M. Deshouillers, F. Hennecart, B. Landreau, *Sums of powers : an arithmetic refinement to the probabilistic model of Erdös and Rényi*, Acta Arithmetica 85, n° 1, 1998.

[54] J.-M. Deshouillers, F. Hennecart, B. Landreau, *Do sums of* 4 *biquadrates have a positive density ?*, ANTS III, Portland, LNCS 1423, 1998.

```
> p[1] := 100996972469714247637786655587969840329509324689190041803603417758904341703348882159067229719:
> for i from 2 to 10 do p[i]:=nextprime(p[i-1]) od:
> seq(p[i+1]-p[i], i=1..9);
            210, 210, 210, 210, 210, 210, 210, 210, 210
```

Pour cela, nous avons repris la méthode mise au point par Harvey Dubner et Harry Nelson en 1997, qui leur avait permis de trouver une progression de sept nombres premiers.[55] L'idée de base consiste à chercher le premier nombre p_1 de la forme

$$p_1 = x + Nm$$

où x et m sont fixés, et N est un entier positif ou nul que l'on fait varier jusqu'à trouver une solution. L'entier m est le produit de petits nombres premiers q_j, et x est choisi de telle sorte qu'aucun des entiers $p_1, p_1 + a, ..., p_1 + (k-1)a$ ne soit divisible par q_j, et que le plus possible de nombres intermédiaires le soient. (La différence commune a est choisie la plus petite possible, ce qui maximise la probabilité de succés, soit $a = 210$ pour $7 \leq k \leq 10$.) Dubner et Nelson ont montré dans leur article de 1997 qu'un tel x peut se trouver facilement par une procédure récursive de recherche. Ma contribution a été principalement d'améliorer cette phase initiale de choix de m et x en incorporant des choix aléatoires à la *Las Vegas*, et y consacrant plus de temps de calcul. En effet, une meilleure valeur de x augmente la probabilité de trouver une solution, donc diminue l'espérance du nombre de valeurs de N à essayer. Toutes proportions gardées, il y a là une certaine analogie avec les améliorations effectuées récemment par Brian Murphy dans la phase de sélection du crible algébrique pour la factorisation entière, où l'on doit trouver des polynômes ayant les meilleures propriétés possibles.

Perspectives. Une progression de 11 nombres premiers en progression arithmétique nécessite une différence commune d'au moins 2310, soit 23090 nombres intermédiaires qui doivent être composés. Le temps de recherche d'une telle progression selon la même méthode est estimé à environ 10^{12} fois celui pour $k = 10$! Il est donc très probable que le record de 10 nombres premiers demeure un certain nombre d'années.

Un problème connexe consiste à trouver la *plus petite* progression de k nombres premiers consécutifs. Pour $k = 6$, on sait[56] depuis 1967 que la plus petite progression commence par 121174811. Pour $k = 7$, nous avons trouvé dans [10] une progression d'entiers de 37 chiffres, mais la plus petite progression reste inconnue ; on peut conjecturer que celle-ci se situe aux alentours de 20 chiffres, donc à la portée d'une recherche exhaustive. En revanche $k = 8$, avec un minimum estimé à 25 chiffres, semble hors de portée.

4.4 Factorisation d'entiers

Grâce à mes contacts avec la communauté internationale de recherche en théorie algorithmique des nombres, j'ai pu participer aux factorisations de RSA-140, achevée le 2

[55] H. Dubner, H. Nelson, *Seven Consecutive Primes In Arithmetic Progression*, Mathematics of Computation, vol. 60, n° 220, 1997.

[56] L. J. Lander, T. R. Parkin, *Consecutive Primes in Arithmetic Progression*, Mathematics of Computation 21, 1967.

février 1999 [5], et de RSA-155, achevée le 22 août 1999 [6], la factorisation de RSA-140 ayant été en quelque sorte une « mise en jambes » pour celle de RSA-155 :

$$\begin{aligned}\text{RSA-155} &= 109417386415705274218097073220403576120037329454492059909138421314763499842889\backslash\\ &\ 34784717997257891267332497625752899781833797076537244027146743531593354333897\\ &= 102639592829741105772054196573991675900716567808038066803341933521790711307779\\ &\times 106603488380168454820927220360012878679207958575989291522270608237193062808643.\end{aligned}$$

Ces deux records de factorisation de nombres « généraux » furent établis grâce au « crible algébrique » (*Number Field Sieve* ou plus simplement NFS). Ma contribution se borna à la première étape de la factorisation, celle de crible à proprement parler. La factorisation de RSA-155 utilisa environ 8400 années-MIPS[57] de temps de calcul, soit seulement quatre fois plus que celle de RSA-140, grâce notamment à un nouvel algorithme dû à Brian Murphy, mis en œuvre par Peter Montgomery, pour la recherche de « bons » polynômes pour le crible algébrique. L'étape de crible, la plus coûteuse, mobilisa 300 machines pendant presque 4 mois ; ma modeste contribution produisit environ 4.5% — soit 5.64 millions — du nombre total de relations.

Au-delà du record, ces calculs ont permis de juger de la sécurité des systèmes cryptographiques basés sur la difficulté de la factorisation entière, en particulier le système RSA. La factorisation de RSA-155 eut un certain retentissement, puisque la barrière psychologique de 512 bits venait d'être franchie.

Richard Brent remarque[58] la relation

$$Y = 13.24 D^{1/3} + 1928.6$$

pour l'année Y où l'on a pu factoriser des entiers quelconques de D chiffres. Si cette formule reste valide dans le futur, on devrait pouvoir casser un nombre de 768 bits (231 chiffres) vers 2010, et un nombre de 1024 bits (309 chiffres) vers 2018. La société RSA a d'ailleurs récemment annoncé[59] des prix en espèces sonnantes et trébuchantes pour la factorisation d'entiers de 576 bits (10000 \$) à 2048 bits (200000 \$) :

$$\begin{aligned}&\text{RSA576} = 188198812920607963838697239461650439807163563379417382700763356422988859715234665\backslash\\ &4853190606065047430453173880113033967161996923212057340318795506569962213051687593076502570 59\ ;\end{aligned}$$

Selon la formule de Brent, ces nombres devraient être décomposés en 2003 pour RSA576 (174 chiffres), et 2041 pour RSA2048 (617 chiffres).

4.5 Trinômes irréductibles

Contexte. Un polynôme $f(x)$ de degré r sur GF(2) est dit *primitif* s'il est irréductible et si x engendre GF(2)/f, c'est-à-dire si $\{x^i, 1 \le i < 2^r\}$ décrit les $2^r - 1$ polynômes non nuls de GF(2)/f. Par exemple, $f = x^6 + x^3 + 1$, qui est irréductible sur GF(2), n'est pas

[57] Une année-MIPS (*MIPS year*) correspond à un an de calcul sur une machine effectuant un million d'instructions par seconde.

[58] R. P. Brent, *Some parallel algorithms for integer factorisation*, EuroPar'99, LNCS 1685, 1999.

[59] `http://www.rsasecurity.com/rsalabs/challenges/factoring/index.html`

primitif, car $x^9 = 1 \bmod x^6 + x^3 + 1$, et x n'engendre que 9 des 63 éléments non nuls de $\mathrm{GF}(2)/f$. En revanche $x^6 + x + 1$ est primitif sur GF(2).

Un polynôme primitif permet de fabriquer un générateur aléatoire de période r. Pour $2^r - 1$ premier, cela revient à chercher des trinômes $x^r + x^s + 1$ irréductibles sur GF(2) (ici et dans la suite, on considère $r > s > 0$). Les nombres premiers de la forme $2^r - 1$ sont appelés nombres de Mersenne ; on en connaît actuellement 38, d'exposants :

$$2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593.$$

Pour obtenir des générateurs aléatoires de grande période, il est donc naturel de choisir r parmi les exposants de Mersenne.

Contribution. Avec Richard Brent et Samuli Larvala, nous avons entrepris mi-2000 la recherche de polynômes de la forme $x^r + x^s + 1$ primitifs sur GF(2). Après avoir vérifié et corrigé[60] les calculs effectués par Kumada *et al.* pour $r = 859433$, nous avons effectué une recherche complète pour $r = 3021377$ [4], et débuté la recherche pour $r = 6972593$.[61] Pour $r = 3021377$, le calcul s'est achevé début 2001, avec deux trinômes primitifs découverts ($s = 361604$ et $s = 1010202$) modulo la transformation $s \to r - s$.

Perspectives. Le cas $r = 6972593$ est un véritable « calcul de l'extrême », avec une estimation de 3 millions d'heures de calcul. D'ailleurs notre demande de ressources en 2001 au CINES (un tiers du temps estimé, soit un million d'heures) changea l'ordinaire des membres du conseil scientifique de cet organisme, puisque même les physiciens les plus gourmands n'avaient demandé que 300000 heures ! Une fois ce calcul terminé, il faudra attendre le prochain nombre de Mersenne trouvé par GIMPS,[62] puisque 6972593 est le dernier connu.

---

[60] Kumada *et al.* avaient en effet « manqué » le trinôme $x^{859433} + x^{170340} + 1$, à cause d'un « bug » dans leur programme : T. Kumada, H. Leeb, Y. Kurita, M. Matsumoto, *New primitive t-nomials* ($t = 3$, 5) *over* GF(2) *whose degree is a Mersenne exponent*, Mathematics of Computation 69, 2000.

[61] Cf `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/trinom.html` pour suivre l'état actuel de nos recherches.

[62] The Great Internet Mersenne Prime Search, `http://www.mersenne.org/`

Et demain ?

Une des particularités de mes travaux de recherche est le fait qu'ils sont indissociables du développement de logiciels (GFUN, MPFR, EPELLE, ...). Ce goût pour ce que d'aucuns appellent les « mathématiques expérimentales » ne devrait pas varier dans les années à venir. Le développement logiciel non seulement valide de nouveaux algorithmes, mais aussi aide à les rendre populaires. D'autre part, la résolution systématique de certaines classes de problèmes grâce à ces logiciels permet d'accéder à de nouvelles questions, et donc pose de nouveaux défis scientifiques.

Mes recherches dans les prochaines années devraient se concentrer sur le thème « arithmétique », avec bien sûr la poursuite du développement de la bibliothèque MPFR et la description des algorithmes qui y sont implantés, à la fois pour certifier la correction de ces algorithmes, et pour faire une synthèse des méthodes d'évaluation de fonctions élémentaires et spéciales. Ma formation en algorithmique et en calcul formel restera sans nul doute extrêmement précieuse !

Pour atteindre l'objectif final de fournir aux logiciels de calcul formel une arithmétique flottante portable, efficace et clairement spécifiée, beaucoup reste à faire. Il faudra en effet pouvoir calculer aussi avec des nombres complexes, et donc définir la notion d'arrondi exact sur les complexes... On pourrait également étendre la notion d'arrondi exact à l'intégration numérique, à la résolution numérique d'équations différentielles, ..., bref assez de sujets passionnants pour au moins 10 autres années de recherche !

Bibliographie

[1] ABBOTT, J., SHOUP, V., AND ZIMMERMANN, P. Factorization in $\mathbb{Z}[x]$: the searching phase. In *Proceedings of ISSAC'2000* (2000), C. Traverso, Ed., ACM Press, pp. 1–7.

[2] BERTAULT, F., RAMARÉ, O., AND ZIMMERMANN, P. On sums of seven cubes. *Mathematics of Computation 68*, 227 (1999), 1303–1310.

[3] BERTAULT, F., AND ZIMMERMANN, P. Unranking of unlabelled decomposable structures. In *Proceedings of Ordal'99, Montpellier* (1999).

[4] BRENT, R. P., LARVALA, S., AND ZIMMERMANN, P. A fast algorithm for testing irreducibility of trinomials mod 2. Rapport technique PRG-TR-13-00, Oxford University Computing Laboratory, 2000. 13 pages.

[5] CAVALLAR, S., DODSON, B., LENSTRA, A., LEYLAND, P., LIOEN, W., MONTGOMERY, P., MURPHY, B., TE RIELE, H., AND ZIMMERMANN, P. Factorization of RSA-140 using the number field sieve. In *Advances in Cryptology, Asiacrypt'99* (Berlin, 1999), L. K. Yan, E. Okamoto, and X. Chaoping, Eds., vol. 1716 of *Lecture Notes in Computer Science*, Springer, pp. 195–207.

[6] CAVALLAR, S., DODSON, B., LENSTRA, A. K., LIOEN, W., MONTGOMERY, P. L., MURPHY, B., TE RIELE, H., AARDAL, K., GILCHRIST, J., GUILLERM, G., LEYLAND, P., MARCHAND, J., MORAIN, F., MUFFETT, A., PUTNAM, C., PUTNAM, C., AND ZIMMERMANN, P. Factorization of a 512–bit RSA modulus. In *Proceedings of Eurocrypt'2000* (Bruges, 2000), B. Preneel, Ed., no. 1807 in Lecture Notes in Computer Science, pp. 1–18.

[7] DANEY, D., HANROT, G., LEFÈVRE, V., ROUILLIER, F., AND ZIMMERMANN, P. The MPFR library. `http://www.mpfr.org/`.

[8] DENISE, A., DUTOUR, I., AND ZIMMERMANN, P. Cs : a MuPAD package for counting and randomly generating combinatorial structures. In *Proceedings of the 10th conference on Formal Power Series and Algebraic Combinatorics* (Toronto, 1998), Fields Institute, pp. 195–204.

[9] DENISE, A., AND ZIMMERMANN, P. Uniform random generation of decomposable structures using floating-point arithmetic. *Theoretical Comput. Sci. 218*, 2 (1999), 219–232.

[10] DUBNER, H., FORBES, T., LYGEROS, N., MIZONY, M., NELSON, H., AND ZIMMERMANN, P. Ten consecutive primes in arithmetic progression. *Mathematics of Computation*. À paraître.

[11] DUTOUR, I., HABSIEGER, L., AND ZIMMERMANN, P. Estimations asymptotiques du nombre de chemins Nord-Est de pente fixée et de largeur bornée. Rapport de recherche RR-3585, Institut National de Recherche en Informatique et en Automatique, 1998.

[12] FLAJOLET, P., SALVY, B., AND ZIMMERMANN, P. Lambda–Upsilon–Omega : An Assistant Algorithms Analyzer. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes* (1989), T. Mora, Ed., vol. 357 of *Lecture Notes in Computer Science*, pp. 201–212.

[13] Flajolet, P., Salvy, B., and Zimmermann, P. Lambda–Upsilon–Omega : The 1989 Cookbook. Rapport de recherche 1073, Institut National de Recherche en Informatique et en Automatique, 1989. 116 pages.

[14] Flajolet, P., Salvy, B., and Zimmermann, P. Automatic Average–case Analysis of Algorithms. *Theoretical Comput. Sci. 79*, 1 (1991), 37–109.

[15] Flajolet, P., Zimmermann, P., and Cutsem, B. V. A calculus for the random generation of labelled combinatorial structures. *Theoretical Comput. Sci. 132*, 1-2 (1994), 1–35.

[16] Fuchssteiner, B., Drescher, K., Kemper, A., Kluge, O., Morisse, K., Naundorf, H., Oevel, G., Postel, F., Schulze, T., Siek, G., Sorgatz, A., Wiwianka, W., and Zimmermann, P. *MuPAD User's Manual.* Wiley Ltd., 1996.

[17] Gomez, C., Salvy, B., and Zimmermann, P. *Calcul formel : mode d'emploi. Exemples en Maple.* Masson, 1995.

[18] Hanrot, G., Quercia, M., and Zimmermann, P. Speeding up the division and square root of power series. Rapport de recherche 3973, Institut National de Recherche en Informatique et en Automatique, 2000.

[19] Hanrot, G., Rivat, J., Tenenbaum, G., and Zimmermann, P. Density results on floating-point invertible numbers. *Theoretical Comput. Sci.* (2001). À paraître.

[20] Louchard, G., Schott, R., Tolley, M., and Zimmermann, P. Random walks, heat equation and distributed algorithms. *Journal of Computational and Applied Mathematics 53* (1994), 243–274.

[21] Postel, F., and Zimmermann, P. Solving ordinary differential equations. In *Computer Algebra Systems : A Practical Guide*, M. Wester, Ed. John Wiley & Sons Ltd, 1999, pp. 191–209.

[22] Rivin, I., Vardi, I., and Zimmermann, P. The n-Queens Problem. *American Mathematical Monthly 101*, 7 (1994), 629–639.

[23] Rouillier, F., and Zimmermann, P. Efficient isolation of a polynomial real roots. Rapport de recherche 4113, Institut National de Recherche en Informatique et en Automatique, 2001. 16 pages.

[24] Salvy, B., and Zimmermann, P. Gfun : A Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Trans. Math. Softw. 20*, 2 (1994), 163–177.

[25] Zimmermann, P. *Séries génératrices et analyse automatique d'algorithmes.* Thèse de doctorat, École Polytechnique, Palaiseau, 1991.

[26] Zimmermann, P. Analysis of functions with a finite number of return values. Rapport de recherche 1625, Institut National de Recherche en Informatique et en Automatique, 1992.

[27] Zimmermann, P. Epelle : un logiciel de détection de fautes d'orthographe. Rapport de recherche 2030, Institut National de Recherche en Informatique et en Automatique, 1993.

[28] Zimmermann, P. Gaïa : a package for the random generation of combinatorial structures. *MapleTech 1*, 1 (1994), 38–46.

[29] Zimmermann, P. Function composition and automatic average case analysis. *Discrete Mathematics 139* (1995), 443–453.

[30] Zimmermann, P. Uniform random generation for the powerset construction. In *Proceedings of the 7th conference on Formal Power Series and Algebraic Combinatorics* (Marne-la-Vallée, 1995), B. Leclerc and J.-Y. Thibon, Eds., pp. 589–600.

[31] Zimmermann, P. *Journées X-UPS 97.* École Polytechnique, Palaiseau, France, 1997, ch. Calcul formel : ce qu'il y a dans la boîte, pp. 47–62.

[32] Zimmermann, P. GMP-ECM : yet another implementation of the Elliptic Curve Method (or how to find a 40-digit prime factor within $2 \cdot 10^{11}$ modular multiplications). Exposé invité au workshop *Computational Number Theory* de FoCM'99, Oxford, 1999.

[33] Zimmermann, P. Karatsuba square root. Rapport de recherche 3805, Institut National de Recherche en Informatique et en Automatique, 1999.

[34] Zimmermann, P. A proof of GMP fast division and square root implementations. `http://www.loria.fr/~zimmerma/papers/`, 2000.

[35] Zimmermann, P. Arithmétique en précision arbitraire. *Calculateurs Parallèles* (2001). À paraître. `http://www.loria.fr/~zimmerma/papers/`.