Dans cette série d'article, je vais présenter quelques aspects de l'implémentation d'un système de calcul formel.

Comme tout programme, un système de calcul formel manipule des données. Une implémentation doit donc d'abord définir des structures de données, ensuite y appliquer des algorithmes efficaces. Les deux aspects sont d'ailleurs liés et il est nécessaire d'introduire des structures de données ad hoc pour traiter certains problèmes. On verra aussi que les problèmes inhérent au calcul formel sont souvent difficiles, leur résolution nécessitant de conjuguer des connaissances en mathématiques et en informatique. C'est d'ailleurs un champ de recherches très actif aujourd'hui.

Ce premier article présente les structures de données utilisées dans les systèmes de calcul formel ; il aura donc une coloration informatique plus marquée que les articles suivants qui présenteront uniquement des algorithmes.

1 Problèmes spécifiques au calcul formel

1.1 Calcul exact et approché, types, évaluation.

Dans les langages de programmation traditionnel (C, Pascal,...), il existe déjà des types permettant une représentation exacte des données (type entier) ou une représentation approchée (type flottant). Mais ces types de donnée de base occupent une taille fixe en mémoire, le types entier est donc limité à un intervalle d'entiers (par exemple $[0, 2^{32} - 1]$ pour un entier non signé sur une machine utilisant un processeur 32 bits) alors que le type flottant peut représenter des nombres réels, mais est limité à une précision en nombre de digits de la mantisse et de l'exposant (par exemple 12 chiffres significatifs et un exposant compris entre -499 et 499).

En calcul formel, on souhaite pouvoir calculer rigoureusement d'une part, et avec des paramètres dont la valeur n'est pas connue d'autre part ; il faut donc s'affranchir de ces limites :

- pour les entiers relatifs, on utilise des entiers de *précision arbitraire* dont la taille en mémoire est dynamique (déterminée pendant l'exécution et non à la compilation),
- pour les nombres complexes, on utilise un couple de nombres réels,
- pour les rationnels, on utilise un couple d'entiers relatifs,
- pour les irrationnels algébriques (par exemple $\sqrt{2}$), on utilise un polynôme irréductible dont ils sont racines,
- pour les paramètres $(x, y, z, t...)$, on utilise un type structuré contenant un champ de type chaine de caractères pour représenter le nom du paramètre et un champ pour attribuer une valeur à (ou une hypothèse sur) ce paramètre,
- pour les nombres transcendants (par exemple π), on est obligé d'introduire un paramètre auquel on attribue une valeur numérique, qui ne sera utilisée qu'au moment où on veut une approximation numérique d'une expression contenant ce nombre transcendant, on parle de constante,

- lorsqu'on a besoin d'une approximation numérique d'un nombre, on peut utiliser des conversions de ces types en un type flottant. On peut aussi pour lutter contre les erreurs d'arrondi utiliser des nombres flottants étendus dont la précision est dynamique ou même des intervalles de flottants étendus,
- il faut aussi un nouveau type, appelé expression ou symbolique, permettant d'appliquer une fonction qu'on ne peut évaluer directement sur les objets précédents, (ou plus généralement une fonction à plusieurs objets de ce type), par exemple $\sin(x)$. Il doit s'agir d'une opération de clôture, au sens où appliquer une fonction à un objet symbolique ne nécessite pas la création d'un nouveau type (en général on renvoie un objet symbolique).

Enfin, il faut pouvoir évaluer un objet (en particulier symbolique) : par exemple évaluer $\sin(x)$ lorsqu'on assigne une valeur à x. Dans cet exemple, on voit qu'il faut d'abord remplacer x par sa valeur avant de lui appliquer la fonction sinus. C'est le mécanisme général de l'évaluation, mais il y a quelques exceptions où on souhaite empêcher l'évaluation d'un ou plusieurs arguments d'une fonction avant l'évaluation de la fonction. Par exemple si on veut calculer la valeur numérique d'une intégrale par des méthodes de quadrature, on ne souhaitera pas rechercher une primitive de la fonction à intégrer. Dans le jargon, on parle alors de "quoter" un argument (l'origine du terme vient probablement de la notation **'** du langage Lisp). Certaines fonctions doivent toujours quoter leurs arguments (par exemple la fonction qui permet de purger le contenu d'un paramètre), on parle alors d'autoquotation.

1.2 Forme normale et reconnaissance du 0.

Une fois défini ces types de base représentant les nombres d'un système de calcul formel, il faut pouvoir comparer ces nombres, en particulier décider si deux représentations distinctes correspondent au même nombre ou, ce qui revient au même, par soustraction décider quand un nombre est nul. Par exemple $4/2$ et 2 représentent le même nombre. Lorsqu'on dispose d'un algorithme permettant de représenter un nombre d'une manière unique, on parle de forme normale. C'est par exemple le cas pour les nombres rationnels, la forme normale usuelle est la fraction irréductible de dénominateur positif. C'est aussi le cas pour les fractions rationnelles de polynômes à coefficients entiers représentées par une fraction irréductible, avec au dénominateur un coefficient de plus haut degré positif. Malheureusement, il n'est pas toujours possible de trouver une forme normale pour diverses raisons théoriques ou pratiques :

- on ne connaît pas toujours le statut de certaines constantes (par exemple la constante d'Euler),
- il n'existe pas d'algorithmes permettant de déterminer s'il existe des relations algébriques entre constantes,
- il n'existe pas forcément une seule forme plus simple, par exemple :

$$\frac{(\sqrt{2}+1)x+1}{x+\sqrt{2}+1} = \frac{x+\sqrt{2}-1}{(\sqrt{2}-1)x+1}$$

Ce cas se présente fréquemment avec les extensions algébriques.

– en pratique il peut être trop coûteux d'utiliser une forme normale, par exemple le polynôme $\frac{x^{1000}-1}{x-1}$ possède 1000 monômes

En résumé, au mieux on a une forme normale, au pire on risque de ne pas reconnaître un zéro, entre les deux on peut ne pas avoir de forme normale mais être capable de reconnaître à coup sûr une expression nulle (par contre, si le système de calcul formel détermine qu'une expression est nulle, alors elle l'est).

On démontre que le problème de la reconnaissance du zéro pour une classe d'expressions "assez générale" est un problème pour lequel il ne peut exister d'algorithme solution. Heureusement, dans la plupart des cas pratiques on sait résoudre ce problème, en se ramenant le plus souvent au cas des polynômes et fractions rationnelles. Par exemple, pour simplifier une expression trigonométrique, on remplace les fonctions trigonométriques $\sin(x), \cos(x), \tan(x)$ par leur expression en fonction de $t = \tan(x/2)$, on est ainsi ramené à une fraction rationnelle en t que l'on écrit sous forme normale.

Les polynômes ont un rôle central dans tout système de calcul formel puisque sauf dans les cas les plus simples (fractions d'entiers par exemple), la simplification d'expressions fait appel à un moment ou à un autre à des calculs de PGCD de polynômes. Le PGCD de polynômes est un algorithme très sollicité auquel nous consacrerons le prochain article. En effet, l'application brutale de l'algorithme d'Euclide pose des problèmes d'efficacité ce qui a obligé à inventer des méthodes plus efficaces. Anticipons rapidement sur un exemple qui montre l'un des problèmes majeurs des algorithmes de calcul formel, l'explosion combinatoire (ici des coefficients des restes successifs). Voici donc les restes successifs lorsqu'on applique l'algorithme d'Euclide pour calculer le PGCD de $P(x) = (x+1)^7 - (x-1)^6$ avec sa dérivée (les deux polynômes sont premiers entre eux) :

$$\begin{array}{r}
7(x+1)^6 - 6(x-1)^5 \\
\frac{162}{49}x^5 + \frac{-390}{49}x^4 + \frac{1060}{49}x^3 + \frac{-780}{49}x^2 + \frac{474}{49}x + \frac{-78}{49} \\
\frac{157780}{729}x^4 + \frac{-507640}{2187}x^3 + \frac{290864}{729}x^2 + \frac{-101528}{729}x + \frac{28028}{729} \\
\frac{1}{49}\left(\frac{1400328}{2645}x^3 + \frac{-732888}{2645}x^2 + \frac{1133352}{3703}x + \frac{-732888}{18515}\right) \\
\frac{1}{2187}\left(\frac{2161816376832}{4669921}x^2 + \frac{-555436846944}{4669921}x + \frac{301917024864}{4669921}\right) \\
\frac{1}{907235}\left(\frac{469345063045455}{129411872}x + \frac{-47641670106615}{129411872}\right) \\
\frac{5497465490623352995840}{209648836272383412129}
\end{array}$$

On peut se convaincre sur d'autres exemples avec le programme `giac` (cf. l'appendice) suivant :

```
pgcd(a):={
  local b,r,res;
```

```
  b:=diff(a,x);
  res:=[];
  for (;b!=0;){
    res:=append(res,b);
    r:=rem(a,b);
    a:=b;
    b:=r;
  }
  return(res);
}
```

1.3 Valeur générique des variables et hypothèses

Lorsqu'on utilise un symbole sans lui affecter de valeurs en mathématiques on s'attend à une discussion en fonction du paramètre représenté par ce symbole. Ce qui nécessiterait de créer un arborescence de calculs (on retrouve ici les problèmes d'explosion combinatoire évoqués dans la section précédente). La plupart des systèmes de calcul formel contournent la difficulté en supposant que le paramètre possède une valeur générique (par exemple la solution de $(t^2-1)x = t-1$ sera $x = 1/(t+1)$) ou choisissent une branche pour les fonctions possédant un point de branchement (par exemple pour résoudre $x^2 = t$ en fonction de t). Certains systèmes demandent de manière interactive à l'utilisateur si la variable est par exemple positive ou différente de 1 mais cela s'oppose à un traitement automatique. On peut aussi anticiper ce type de décision en faisant des hypothèses sur une paramètre, la plupart des systèmes de calcul formel actuel proposent cette possibilité.

2 Structures de données

On a vu plus haut qu'on souhaitait manipuler des entiers de taille non fixe, des réels de précision fixe ou non, des fractions, des nombres complexes, des extensions algébriques, des paramètres, des expressions symboliques. Il faut un type générique recouvrant ces divers types de scalaire.

On peut utiliser un type structuré comportant un champ type et la donnée ou un pointeur sur la donnée (avec dans ce cas un pointeur sur un compteur de références de la donnée pour pouvoir la détruire dès qu'elle n'est plus référencée[1]). En programmation orientée objet, on utiliserait plutôt un type abstrait dont dérivent ces différents scalaires.

Il faut aussi un type pour les vecteurs, les matrices et les listes. On peut se poser la question de savoir s'il faut inclure ces types dans le type générique ; en général la réponse est affirmative, une des raisons étant que les interpréteurs

[1]Les systèmes de calcul formel embarqués (calculatrices) ou devant gérer des calculs intensifs utilisent d'ailleurs des méthodes spécifiques pour gérer le problème de la fragmentation de la mémoire, appelés "garbage collector". Ce type de méthode est intégré dans des langages comme Lisp ou Java, en C/C++ on trouve des libraries pour cela, par exemple GC de Boehm.

qui permettront de lire des données dans un fichier texte sont en général basé sur le couple `lex(flex)/yacc(bison)` qui ne peut compiler qu'à destination d'un seul type. Ceci permet également d'unifier en un seul type symbolique les fonctions ayant un ou plusieurs arguments en voyant plusieurs arguments comme un vecteur d'arguments. Les fonctions sont le plus souvent elle-même incluses dans le type générique permettant ainsi à l'utilisateur de saisir des commandes ou programmes fonctionnels (on peut utiliser une fonction comme argument d'une commande).

Pour des raisons d'efficacité, les systèmes de calcul formel utilisent souvent des représentations particulières pour les polynômes dont on a dit qu'ils jouaient un rôle central. Pour les polynômes à une variable, on peut utiliser la liste des coefficients du polynôme, on parle alors de représentation dense. On peut aussi décider de ne stocker que les coefficients non nuls, on parle alors de représentation creuse (on stocke alors un couple formé par le coefficient et le degré du monôme correspondant). Pour les polynômes à plusieurs variables, on peut les considérer comme des polynômes à une variable à coefficients polynomiaux, on parle alors de représentation récursive. On peut aussi décider de ne pas briser la symétrie entre les variables (pas de variable principale), on parle alors de représentation distribuée, le plus souvent les représentation distribuées sont creuses car les représentations denses nécessitent très vite beaucoup de coefficients. Les méthodes de représentation creuses sont parfois aussi utilisées pour les matrices ayant beaucoup de coefficients nuls.

Voyons maintenant plus précisément sur quelques exemples de logiciels de calcul formel répandus quelles structures de données sont utilisées. Plusieurs éléments entrent en compte dans les choix faits :

- le(s) profil(s) d'utilisation (enseignement, ingéniérie, calcul intensif, recherche)
- les ressources disponibles (mémoire, puissance du processeur...)
- la facilité d'implémentation (choix du langage, outils disponibles en particulier débuggueurs, ...)
- l'histoire du système (un système conçu avec les outils disponibles aujourd'hui est forcément différent d'un système conçu il y a 20 ans)

Nous allons d'abord parler des calculatrices formelles HP et TI (le lecteur pourra facilement les tester grâce aux émulateurs gratuits pour PC). Ce sont des systèmes plutôt destinés à l'enseignement et soumis à de fortes contraintes en termes de taille mémoire, destinés à traiter des petits problèmes. Puis nous présenterons des systèmes pour ordinateur où cet aspect importe beaucoup moins ce qui permet d'utiliser des langages de programmation de plus haut niveau.

2.1 Calculatrices formelles HP

Les langages utilisés pour programmer ces calculateurs sont l'assembleur et le RPL (Reverse Polish Lisp) adapté à l'écriture de code en mémoire morte très compact.

Le type générique est implémenté avec un champ type appelé prologue (qui est en fait un pointeur sur la fonction chargée d'évaluer ce type d'objet) suivi de la donnée elle-même (et non d'un pointeur sur la donnée, on économise ainsi la place mémoire du compteur de référence).

Le type entier en précision arbitraire est codé par le nombre de digits (sur 5 quartets[2]) suivi du signe sur un quartet et de la représentation BCD (en base 10) de la valeur absolue de l'entier. Le choix de la représentation BCD a été fait pour optimiser les temps de conversion en chaîne de caractères pour l'affichage. La mémoire vive disponible est de 256K, c'est elle qui limite la taille des entiers et non le champ longueur de l'entier. Il n'y a pas de type spécifique pour les rationnels (on utilise un objet symbolique normal).

Les fonctions internes des HP49 et HP40 utilisent le type programme pour représenter les entiers de Gauß (complexes dont la partie réelle et imaginaire est entière). Les nombres algébriques ne sont pas implémentés, sauf les racines carrées (représentée de manière interne par le type programme). Il y a un type spécifique prévu pour les flottants en précision arbitraire, mais l'implémentation des opérations sur ces types n'a pas été intégrée en ROM à ce jour.

Les types listes, programmes et objet symbolique sont composés du prologue (champ type) suivi par la succession d'objets situés en mémoire vive ou de pointeurs sur des objets situés en mémoire en lecture seule (ROM) et se terminent par un pointeur sur une adresse fixe (appelée `SEMI`). Ces types sont eux-mêmes des objets et peuvent donc être utilisés de manière récursive. La longueur des types listes, programmes, symboliques n'est stockée nulle part, c'est le délimiteur final qui permet de la connaître, ce qui est parfois source d'inefficacité. On utilise de manière interne les listes pour représenter les polynômes denses (avec représentation récursive pour les polynômes à plusieurs variables).

Les calculatrices HP4xG utilisent une pile[3], c'est-à-dire une liste de taille non fixée d'objets. On place les objets sur la pile, l'exécution d'une fonction prend ces arguments sur la pile et renvoie un ou plusieurs résultats sur la pile (ce qui est une souplesse du RPN comparé aux langages où on ne peut renvoyer qu'une valeur de retour). Il faut donc donner les arguments avant d'appeler la fonction correspondante. Par exemple pour calculer $a+b$ on tapera `a b +`. C'est la syntaxe dite polonaise inversée (RPN). Un avantage de cette syntaxe est que le codage d'un objet symbolique par cette syntaxe est évidente, il suffit de stocker la liste précédente `{a b +}`. Les objets symboliques sont donc représenté par une suite d'objets écrit en syntaxe polonaise inversée. L'évaluation d'un objet symbolique se fait dans l'ordre polonaise inversé : les arguments sont évalués puis les fonctions leur sont appliqués. Pour des raisons d'efficacité, on représente souvent les objets composites (listes, symboliques) par leurs composants placés sur la pile (appelé meta-objets).

Une rigidité de la syntaxe polonaise est que les fonctions ont toujours un nombre fixe d'arguments[4], par exemple l'addition a toujours 2 arguments, ainsi $a+b+c$ est obtenu par $(a+b)+c$ ou par $a+(b+c)$ c'est-à-dire respective-

[2]un quartet=un demi octet

[3]Plus précisément deux piles, la pile de donnée et la pile gérant le flux d'exécution. Cette dernière n'est pas visible par l'utilisateur

[4]Sauf si on utilise comme dernier argument le nombre d'arguments de la fonction ou si on utilise (cf. infra) un tag de début de liste d'arguments

ment `a b + c +` ou `a b c + +` ce qui brise parfois artificiellement la symétrie de certaines opérations. En polonaise inversée, le système doit de plus jongler avec l'autoquote puisque les arguments sont évalués avant l'opérateur qui éventuellement demande à ne pas évaluer ses arguments. L'utilisateur peut quoter une sous-expression.

Les hypothèses sur des variables réelles sont regroupées dans une liste stockée dans la variable globale `REALASSUME`, on peut supposer qu'une variable est dans un intervalle. Il n'y a pas à ce jour de possibilité de supposer qu'une variable est entière (ni à fortiori qu'une variable à une valeur modulo un entier fixé), bien qu'il ait été décidé de réserver la variable globale `INTEGERASSUME` à cet effet. Il n'y a pas de possibilité de faire des hypothèses ayant une portée locale.

2.2 Calculatrices formelles TI

Le langage utilisé pour programmer ces calculatrices est le langage C (on peut aussi bien sur écrire du code en assembleur). On retrouve ici les différents types de données regroupé en un type générique qui est un tableau d'octets (aussi appelé quantum). Le champ type est appelé tag dans la documentation TI. Contrairement à ce qui précède, ce champ type est placé en mémoire à la fin de l'objet, ce qui est possible car la longueur d'un objet est toujours indiquée au début de l'objet. Ceci est fait afin de faciliter l'évaluation (cf. infra).

Les entiers en précision arbitraire sont codé par deux tags possibles (pour différencier le signe), un octet pour la longueur, puis la valeur absolue de l'entier (en base 256). Ils sont donc limités par le champ longueur à 255 octets, le plus grand entier représentable est $(256^{255}-1)$[5]. Il existe un tag spécifique pour les rationnels, pour les constantes réelles et entières qui apparaissent par exemple en résolvant une équation. Il existe des tags utilisés de manière interne, par exemple pour les nombres complexes. Il n'y a pas de tag prévu pour les flottants en précision arbitraire. ni pour les nombres algébriques (racines carrées par exemple).

Les listes sont codées par la suite de leurs éléments. En principe elles ne peuvent pas contenir des listes sauf si ce sont des matrices. Quelques fonctions utilisent les listes pour représenter des polynômes denses à une variable, mais probablement pas pour représenter de manière récursive des polynômes à plusieurs variables (puisque le type liste n'est en principe pas récursif).

Comme sur les HP, les TI utilisent une pile (non visible par l'utilisateur) appelée expression stack afin de traduire un expression mathématique sous forme d'un texte en un objet symbolique codé exactement comme ci-dessus en syntaxe polonaise. Toutefois, la présence du champ longueur permet d'évaluer un objet symbolique sans perdre en efficacité en partant de l'opérateur final et en redescendant ensuite sur ces arguments, c'est la stratégie adoptée par ces calculatrices. C'est pour cela que le tag d'identification se trouve à la fin de l'objet. L'utilisation de cette méthode facilite grandement l'autoquotation (on

[5]Toutefois une adaptation du logiciel utilisant comme quantum de base par exemple 32 bits porterait cette limite à $65536^{65535}-1$

peut toutefois regretter que le système n'ait pas prévu d'instruction permettant à l'utilisateur d'empêcher l'évaluation d'une sous-expression).

On ne peut pas faire d'hypothèse globale sur un paramètre par contre on peut faire des hypothèses de type appartenance à un intervalle ayant une portée locale.

2.3 Maple, MuPAD, Mathematica, ...

Ces systèmes ont un noyau fermé, au sens où l'utilisateur n'a pas accès du tout, ou en tout cas pas facilement, aux structures de données de base. Je ne dispose donc pas d'information sur les structures de données utilisées par le noyau (pour MuPAD, on pourrait sans doute en savoir plus en achetant de la documentation sur la programmation des modules dynamiques).

L'interaction système-utilisateur se fait quasiment toujours en utilisant le langage de programmation propre au système, langage interprété par le noyau du système (ce qui ralentit l'exécution). Ces langages utilisateurs sont essentiellement non typés : on travaille avec des variables du type générique sans pouvoir accéder aux types sous-jacents. On ne bénéficie donc pas des vérifications faites lors de la compilation avec un langage typé, de plus ces systèmes ne sont pas toujours fourni avec de bon outils de mise au point. Enfin ces langages ne sont pas standardisés d'un système à l'autre et il est en général impossible d'utiliser ces systèmes comme des librairies depuis un langage de programmation traditionnel. Leur intérêt principal réside donc dans une utilisation interactive en profitant de la librairie de fonctions accessibles.

2.4 Giac/xcas

Il s'agit du système de calcul formel que j'implémente actuellement sous forme d'une bibliothèque C++ (ce qui permettra aux programmes tiers d'utiliser beaucoup plus facilement du calcul formel qu'avec les systèmes précédents). L'objectif est d'avoir un système facile à programmer directement en C++, proche du langage utilisateur, lui-même compatible avec Maple ou MuPAD, tout cela sans trop perdre en performances comparativement aux librairies spécialisées écrites en C/C++. Ce qui explique un choix de type générique (`gen`) non orienté objet, avec un champ type et soit une donnée immédiate pour les nombres flottants par exemple, soit un pointeur vers un objet du type correspondant au champ type pour les données de taille non fixe (on pourrait donc se contenter du langage C, mais le langage C++ permet de redéfinir les opérateurs sur des types utilisateurs ce qui améliore considérablement la lisibilité du code source). Les données dynamiques ne sont pas dupliquées, Giac utilise un pointeur sur un compteur de référence pour détruire ces données lorsqu'elles ne sont plus référencées.

Les entiers en précision arbitraire sont hérités de la bibliothq̀ue GMP (écrite en C) du projet GNU. Les flottants en précision arbitraire utiliseront aussi GMP (plus précisément MPFR). Il y a un type fraction, structure C composé d'un champ numérateur et d'un champ dénominateur, et un type nombre complexe.

Les listes, vecteurs, matrices utilisent le type paramétré `vector<>` de la librairie standard C++ (Standard Template Library). Les objets symboliques sont des structures composés d'un champ sommet qui est une fonction prenant un argument de type `gen` et renvoyant un résultat de type `gen`, et d'un champ feuille qui est de type `gen`. Lorsqu'une fonction possède plusieurs arguments, ils sont rassemblés en une liste formant la feuille de l'objet symbolique. Les programmes sont aussi des objets symboliques, dont le champ sommet est la fonction évaluation d'un programme. Les listes sont aussi utilisées pour représenter vecteur, matrices et polynômes en une variable en représentation dense. Les polynômes en représentation creuse ou en plusieurs indéterminées sont également disponibles.

L'évaluation d'un objet symbolique se fait en regardant d'abord si la fonction au sommet doit évaluer ou non ses arguments (autoquote), on évalue les arguments si nécessaire puis on applique la fonction.

Une hypthèse sur un paramètre est simplement une valeur spéciale affectée au paramètre, valeur ignorée par la routine d'évaluation.

3 Algorithmes et complexité.

On va présenter dans la suite quelques algorithmes que l'on peut considérer comme classiques dans le domaine du calcul formel. Avant d'implémenter ce type d'algorithmes, on a besoin des algorithmes de base en arithmétique. Le lecteur trouvera en appendice une brève présentation de certains de ces algorithmes, mes références en la matière sont le livre de Henri Cohen, et les livres de Donald Knuth (cf. appendice).

La plupart des problèmes posés en calcul formel nécessitent des calculs dont la taille croit de manière exponentielle voire doublement exponentielle en fonction de la taille des données et ce même si le résultat est lui aussi de taille petite. Un exemple est la réduction des systèmes de plusieurs équations polynomiales (bases de Groebner). Dans certains cas, l'application de théories mathématiques parfois sophistiquées permet de réduire la complexité (par exemple, M. Van Hoeij a découvert récemment qu'un algorithme très utilisé en théorie des nombres, l'algorithme LLL, permettait d'améliorer la complexité d'une des étapes de la factorisation des polynomes à coefficients entiers sur les entiers). Heureusement, dans de nombreux cas, on peut réduire la complexité (donc le temps de calcul) par des adaptations au problème d'une même idée à condition de faire des hypothèses sur les données (autrement dit en abandonnant la volonté d'implémenter un algorithme générique).

Par exemple lorsqu'on travaille avec des entiers (ou des polynômes à coefficients entiers, ou des matrices à coefficients entiers...) on utilise souvent des algorithmes modulaires et p-adiques. Comme le calcul exact nécessite presque toujours de calculer avec des entiers, ces méthodes ont un rôle central en calcul formel, nous les présentons donc maintenant brièvement. Dans les prochains articles, nous utiliserons ce type de méthode, par exemple pour le calcul de PGCD ou la factorisation de polynômes à coefficients entiers.

Les méthodes modulaires consistent à réduire un problème dans $\mathbb{Z}$ à son équivalent dans $Z/n\mathbb{Z}$ pour une ou plusieurs valeurs de n, nombre premier. Le calcul dans $\mathbb{Z}/n\mathbb{Z}$ a l'avantage de se faire avec des entiers dont la taille est bornée. Ensuite à l'aide d'estimations à priori sur la taille des solutions éventuelles du problème initial, on reconstruit la solution au problème initial avec le théorème des restes chinois.

Par exemple, on peut calculer un déterminant d'une matrice à coefficients entiers en cherchant ce déterminant dans $\mathbb{Z}/n\mathbb{Z}$ pour plusieurs premiers n, dont le produit est plus grand qu'une estimation à priori de la taille du déterminant (donnée par exemple par l'inégalité d'Hadamard).

Les méthodes p-adiques commencent de manière identique par un calcul dans $\mathbb{Z}/n\mathbb{Z}$, on augmente ensuite la précision de la solution en la « liftant » de $\mathbb{Z}/n^k\mathbb{Z}$ vers $\mathbb{Z}/n^{k+1}\mathbb{Z}$ ou vers $\mathbb{Z}/n^{2k}\mathbb{Z}$ (lift linéaire ou lift quadratique), on s'arrête lorsque k est assez grand (à l'aide d'estimations à priori) et on reconstruit alors la solution initiale. L'étape de « lift » est en général un "lemme de Hensel"dont on verra quelques exemples dans les prochains articles. L'algorithme commun au lemme de Hensel et au théorème des restes chinois est l'identité de Bézout, que l'on retrouve d'ailleurs un peu partout (par exemple pour le calcul de primitives).

Illustrons cette méthode sur un exemple simple, la recherche de racines rationnelles d'un polynôme $P(X) = a_nX^n + \cdots + a_0$ à coefficients entiers ou polynomiaux, avec a_n et a_0 non nuls. L'algorithme générique (assez connu) consiste à chercher les diviseurs de a_0 et de a_n et à tester toutes les fractions de ces diviseurs, on montre en effet aisément que si $X = p/q$ fraction irréductible est racine de P alors q divise a_n et p divise a_0. Cet algorithme est très inefficace si a_n ou a_0 est un grand entier (car on ne sait pas forcément le factoriser) ou s'il a beaucoup de facteurs premiers (la liste des diviseurs à tester est alors très grande).

Lorsque les coefficients de P sont entiers, la recherche précédente revient à trouver un facteur à coefficients entiers $qX - p$ de P on peut donc réduire le problème modulo un entier premier n qui ne divise pas a_n : si un tel facteur existe dans $\mathbb{Z}$ alors ce facteur (réduit modulo n) est un facteur de P dans $\mathbb{Z}/n\mathbb{Z}$ donc P admet une racine dans $\mathbb{Z}/n\mathbb{Z}$ (puisque q est inversible modulo n car on a choisi n premier ne divisant pas a_n). On évalue maintenant P en les n éléments de $\mathbb{Z}/n\mathbb{Z}$. S'il n'y a pas de 0, alors P n'a pas de racine rationnelle. S'il y a des racines, on va les lifter de $\mathbb{Z}/n^k\mathbb{Z}$ dans $\mathbb{Z}/n^{2k}\mathbb{Z}$.

On suppose donc que pour $k \geq 1$, il existe un entier p_k tel que

$$P(p_k) = 0 \pmod{n^k}$$

Il s'agit de trouver un entier x tel que $p_{k+1} = p_k + n^k \times x$ vérifie

$$P(p_{k+1}) = 0 \pmod{n^{2k}}$$

On applique la formule de Taylor à l'ordre 1 pour P en p_k, le reste est nul modulo n^{2k}, donc :

$$P(p_k) + n^k x P'(p_k) = 0 \pmod{n^{2k}}$$

soit finalement :
$$x = -\frac{P(p_k)}{n^k}(P'(p_k) \pmod{n^k})^{-1}$$
On reconnaît au passage la méthode de Newton, pour qu'elle fonctionne il suffit que $P'(p_k) \neq 0 \pmod n$ ce qui permet de l'inverser modulo n^k (et c'est ici qu'intervient l'identité de Bézout). En pratique quand on factorise un polynôme, on commence par retirer les multiplicités, on peut donc supposer que P est sans facteur multiple dans $\mathbb{Z}$. Ceci n'entraîne pas forcément qu'il le reste dans $\mathbb{Z}/n\mathbb{Z}$ ce qui crée une contrainte supplémentaire sur le choix de n, à savoir que P et P' restent premier entre eux dans $\mathbb{Z}/n\mathbb{Z}$ (il existe forcément de tels n, par exemple n premier plus grand que le plus grand entier intervenant dans le calcul du PGCD de P et P' dans $\mathbb{Z}$).

Reste donc à revenir dans $\mathbb{Z}$ à partir d'une racine p_k dans $\mathbb{Z}/(n^k\mathbb{Z})$ (où on peut choisir k). On va maintenant utiliser la représentation modulaire symétrique : on prend comme représentant modulaire d'un entier z dans $\mathbb{Z}/n^k\mathbb{Z}$ l'unique entier congru à z modulo n qui est strictement compris entre $-n^k/2$ et $n^k/2$ (si n est pair, la deuxième inégalité est large).

Si $qX - p$ est un facteur de P, alors $a_nX - \frac{a_n}{q}p$ est encore un facteur de P (le quotient de P par $a_nX - \frac{a_n}{q}p$ est à coefficients rationnels mais le facteur est à coefficients entiers). Si on a choisi k tel que $n^k > 2|a_na_0|$, l'écriture en représentation modulaire symétrique de $a_nX - \frac{a_n}{q}p$ est inchangée, en effet on a des estimations à priori sur les entiers p et q : $|q| \leq |a_n|$ et $|p| \leq |a_0|$ puisque q divise a_n et p divise a_0. Comme $a_nX - \frac{a_n}{q}p$ est égal à $a_n(X-p_k)$ dans $\mathbb{Z}/(n^k\mathbb{Z})$, il nous suffit d'écrire en représentation modulaire symétrique $a_n(X - p_k) = a_nX - p'$. Pour conclure, on sait que $a_nX - p'$ est un multiple entier de $qX - p$. On divise donc le facteur a_nX-p' par le pgcd de a_n et p' et on teste la divisibilité de P par ce facteur réduit.

Exemple

Considérons le polynôme $2X^3 - X^2 - X - 3$ qui est sans facteur carré. On ne peut pas choisir $n = 2$ car on réduirait le degré, pour $n = 3$, on a $P' = X - 1$ qui est facteur de P, pour $n = 5$, $P' = 6X^2-2X-1$, on vérifie que P et P' sont premiers entre eux (par exemple avec `MODGCD` sur une HP49 où on aura fixé la variable `MODULO` à 5).

On teste ensuite les entiers de -2 à 2 sur P. Seul -1 est racine modulo 5 ($P(-1) = -5$), on va maintenant lifter $p_1 = -1$.

L'estimation à priori est $2|a_n||a_0| = 12$ donc $k = 2$ ($5^2 = 25 > 12$), une itération suffira. On a $P'(-1) = 7$, l'inverse de $P'(-1) \pmod 5$ est -2 donc :
$$x = -\frac{P(-1)}{5}(-2) = -(-1)(-2) = -2$$
et $p_2 = -1 + 5 \times (-2) = -11$ est racine de P dans $\mathbb{Z}/25\mathbb{Z}$. On calcule ensuite $a_n(X - p_k) = 2(X + 11) = 2X + 22 = 2X - 3$ en représentation symétrique, le PGCD de 2 et -3 est 1 donc on teste le facteur $2X - 3$, ici il divise P donc P admet un unique facteur entier de degré 1 qui est $2X - 3$.

A Quelques algorithmes fondamentaux en arithmétique.

- Les algorithmes de multiplication et division dit rapides des entiers et polynômes (Karatsuba, FFT, ...). Cf. par exemple Knuth. ou pour les entiers la documentation de GMP.
- Au lieu de la division euclidienne, on utilise très souvent la pseudo-division pour les polynômes : étant donné deux polynômes A et B de degrés a et b à coefficients dans un anneau contenu dans un corps (par exemple $\mathbb{Z}$), on multiplie A par une puissance du coefficient dominant B_b de B, plus précisément par B_b^{a-b+1}, ce qui permet d'effectuer la division par B sans que les coefficients sortent de l'anneau.
$$B_b^{a-b+1}A = BQ + R$$
On utilise cette méthode lorsqu'on peut multiplier les polynômes par des constantes sans changer le problème (par exemple pour l'algorithme d'Euclide).
- L'algorithme d'Euclide est un algorithme « générique » de calcul de PGCD. Il n'est en général pas utilisé tel quel. Pour les entiers on utilise une variation adaptée à la représentation binaire des entiers (cf. Cohen ou le manuel de GMP version 4 pour plus de détails). Nous décrirons des algorithmes de PGCD plus efficaces pour les polynômes dans le prochain article.
- l'identité de Bézout, aussi appelée PGCD étendu. Étant donné deux entiers ou deux polynômes a et b on calcule u, v et d tels que $au + bv = d$. On écrit la matrice :
$$\begin{pmatrix} a & 1 & 0 \\ b & 0 & 1 \end{pmatrix}$$
où on remarque que pour chaque ligne le coefficient de la 1ère colonne est égal à a multiplié par le coefficient de la 2ème colonne additionné à b multiplié par le coefficient de la 3ème colonne. Ce qui reste vrai si on effectue des combinaisons linéaires de lignes (type réduction de Gauß). Comme on travaille dans les entiers ou les polynômes, on remplace la réduction de Gauß des matrices à coefficients réels par une combinaison linéaire utilisant le quotient *euclidien* q de a par b. On obtient alors le reste r en 1ère colonne :
$$L_3 = L_1 - qL_2 \quad \begin{pmatrix} a & 1 & 0 \\ b & 0 & 1 \\ r & 1 & -q \end{pmatrix}$$
et on recommence jusqu'à obtenir 0 en 1ère colonne. L'avant-dernière ligne obtenue est l'identité de Bézout (la dernière ligne donne le PPCM de a et b). Si l'on veut l'inverse de a modulo b on remarque qu'il n'est pas utile de calculer les coefficients appartenant à la 3ème colonne. Enfin, les lignes intermédiaires peuvent servir à reconstruire une fraction d'entier représentée par un entier de $\mathbb{Z}/n\mathbb{Z}$ lorsque le numérateur et le dénominateur sont de valeur absolue inférieure à $\sqrt{n/2}$.

– Le théorème des restes chinois. Si on connaît $x = a \pmod{m}$ et $x = b \pmod{n}$ avec m et n premiers entre eux, on détermine c tel que $x = c \pmod{m \times n}$ ($c = a + mu = b + nv$ et on applique Bézout pour trouver u et v).
– Les tests de pseudo-primalité. Il est essentiel d'avoir une méthode rapide permettant de générer des nombres premiers pour appliquer des méthodes modulaires et p-adiques. On utilise par exemple le test de Miller-Rabin, qui prolonge le petit théorème de Fermat (si p est premier, alors $a^p = a \pmod{p}$).

B Pour en savoir plus.

Sur des aspects plus théoriques :
– Knuth : TAOCP (The Art of Computer Programming), volumes 1 et suivants
– Henri Cohen : A Course in Computational Algebraic Number Theory
– Davenport, Siret, Tournier : Calcul formel : Systèmes et algorithmes de manipulations algébriques

Sur des aspects plus pratiques, quelques références en ligne, la plupart sont accessibles gratuitement :
– le code source de Giac disponible à l'URL :
`http://www-fourier.ujf-grenoble.fr/~parisse/giac.html`
– le code source de GiNaC, cf. : `http://www.ginac.de`
– le site `http://www.hpcalc.org` pour les calculatrices HP, on y trouve tout, de la documentation, des émulateurs de calculatrices HP, des outils de développement pour Windows et Unix/Linux, ... Pour ce qui concerne cet article, je conseille de lire
`http://www.hpcalc.org/hp48/docs/programming/rplman.zip`
– le site `http://www.ticalc.org`, on y trouve le portage tigcc du compilateur C de GNU, des émulateurs, etc. Des informations de cet article ont leur source dans le guide du développeur TI89/92
`http://education.ti.com/developer/8992/download/download.html`
– la librairie du système `MuPAD` (archivée dans le fichier `share/lib.tar` des distributions Unix), on peut se procurer MuPAD par exemple sur :
`ftp://ftp.inria.fr/lang/MuPAD`
cf. `www.sciface.com` pour obtenir une licence d'utilisation.
– pour les personnes ayant accès à Maple, il est possible de décompiler une instruction `Maple` avec la commande
`eval(instruction);`
après avoir tapé
`interface(verboseproc=2);`
– le source du plus ancien système de calcul formel `maxima` (devenu logiciel libre) pour les personnes familières du langage Lisp
`http://www.math.utexas.edu/users/wfs`
– le source de librairies plus spécialisées (GMP, GP-PARI, Singular, NTL,

Zen, ALP, Cocoa, ...), cf. le site Scientific Application for Linux
`http://www-sor.inria.fr/mirrors/sal/index.shtml`