Arithmétique exacte

Paul Zimmermann

Projet Spaces, LORIA/INRIA Lorraine

« *Ce que j'aimerais dans ce cours, ce sont plus les algorithmes qui permettent d'effectuer des opérations exactes que leur utilisation.* »

Exact vs Inexact

Exemple : calcul de $\sqrt{x}$ pour $x = 474256143563$ (12 chiffres).

Méthode « inexacte » :

```
> x:=474256143563:
> Digits:=20: evalf(sqrt(x));
                                688662.57598551120919
```

Méthode « exacte » :

```
> X:=10^12*x: s:=isqrt(X): r:=X-s^2: s, r;
                             688662575986, -673223872196
```

Entiers/flottants vs polynômes/séries

Calculs exacts : entiers $\Longleftrightarrow$ polynômes

Calculs tronqués : flottants $\Longleftrightarrow$ séries de Taylor

Algorithme de Kronecker/Schönhage : polynômes $\rightarrow$ entiers

```
> p := 68*x^5+21*x^4+9*x^3+55*x^2+63*x+57:
> q := 60*x^5+74*x^4+85*x^3+16*x^2+61*x+7:
> sort(expand(p*q));
     10          9          8          7           6           5           4
4080 x   + 6292 x   + 7874 x   + 6839 x   + 13099 x   + 14658 x   + 11149 x

             3          2
     + 9271 x   + 5140 x   + 3918 x + 399
```

$$4080\,x^{10} + 6292\,x^{9} + 7874\,x^{8} + 6839\,x^{7} + 13099\,x^{6} + 14658\,x^{5} + 11149\,x^{4}$$

$$+ 9271\,x^{3} + 5140\,x^{2} + 3918\,x + 399$$

```
> a:=subs(x=100000, p);
                         a := 680002100009000550006300057

> b:=subs(x=100000, q);
                         b := 600007400008500016000610007

> a*b;
              408006292078740683913099146581114909271051400391800399
```

Algorithmes rapides de multiplication par évaluation/interpolation : entiers → polynômes

Avantages du calcul exact

Principe de base : calculer aussi le terme d'erreur (reste) en se ramenant à des calculs entiers.

• le calcul d'arrondi devient facile (ici il suffit de regarder le signe de r pour un arrondi dirigé, ou de comparer $|r|$ à $|s \pm \frac{1}{4}|$ pour l'arrondi au plus proche).

• le calcul lui-même peut être accéléré (typiquement l'itération de Newton commence par re-calculer le terme d'erreur).

• le calcul d'erreur est facilité par les instructions du type fma (*fused multiply add*).

Calcul exact des corrections (1)

Boldo, Daumas, *Representing correcting terms for possibly underflowing floating-point operations*, Arith'16, 2003.

Addition : erreur (arrondi au plus proche) représentable sur n bits.

Algorithme TwoSum (Knuth 1973).

$z \leftarrow x \oplus y$

$y_v \leftarrow z \ominus x$

$x_v \leftarrow z \ominus y_v$

$y_r \leftarrow y \ominus y_v$

$x_r \leftarrow x \ominus x_v$

$\epsilon \leftarrow x_r \oplus y_r$

Prouvé par Shewchuk en base 2 (précision ≥ 3), Priest (avec condition $|a \oplus a| \leq 2|a|$), Knuth (*"very intricate"*).

Calcul exact des corrections (2)

Multiplication : facile avec fma.

$$z \leftarrow x \otimes y$$
$$\epsilon \leftarrow \circ(xy - z).$$

Division : pas toujours possible sous forme $x/y = z + \epsilon$, mais ok avec $x = yz + \epsilon$ (cf. multiplication).

Idem pour racine carrée : $x = s^2 + \epsilon$.

Plan

Au lieu de présenter en détail des algorithmes, on va voir quelques idées (règles) qui ont permis leur invention.

Complexité des algorithmes :

• addition, soustraction, comparaison : $O(n)$

• multiplication : $M(n) = O(n \log n)$ pour les polynômes, $O(n \log n \log \log n)$ pour les entiers

• opérations algébriques : $O(M(n))$ (Newton). Attention aux constantes !

• autres opérations (conversion, pgcd, fonctions élémentaires) : $O(M(n) \log n)$ voire $O(M(n) \log^2 n)$.

Multiplication

- naïfs : $O(n^2)$
- Karatsuba (1962) : $O(n^\alpha)$ avec $\alpha = \log_2 3 \approx 1.585$.
- Toom-Cook r-way : $O(n^\alpha)$ avec $\alpha = \log_r(2r-1) \approx 1.465$ pour $r = 3$, 1.404 pour $r = 4$.

Idée : $A(x) = a_0 + a_1 x + \cdots + a_{r-1}x^{r-1}$,
$B(x) = b_0 + b_1 x + \cdots + b_{r-1}x^{r-1}$, $AB(x) = c_0 + \cdots + c_{2r-1}x^{2r-2}$.

1. Évaluer $A(x), B(x)$ en $2r-1$ points
2. Effectuer (récursivement) les $2r-1$ produits terme à terme
3. Interpoler pour trouver $c_0, \ldots, c_{2r-2}$.

- FFT (1971) : $O(n \log n \log\log n)$

Classes d'algorithmes (opérations algébriques)

- naïfs : $O(n^2)$
- *divide and conquer* : $O(n^\alpha)$ ou $O(M(n)\log n)$
- Newton : $O(M(n))$.

Règle 1 : Tous les coups sont permis

Deux grandes classes d'algorithmes pour les calculs sur les entiers :

- classiques (par les poids forts)
- modulaires (par les poids faibles, p-adiques)

Avantage des algos p-adiques : tout exact (les retenues vont dans le bon sens).

Exemple : division exacte $q = \frac{a}{b}$: $q = \frac{a}{b} \bmod 2^n$ pour $n \geq l(a) - l(b) + 1$ (Jebelean, 1993).

Calcul flottant :

```
> a := 146875534644858625262457:
> b := 343633073697:
> Digits:=12: a1 := evalf(a);
                                                   24
                              a1 := .146875534645 10

> q1 := round(a1/b);
                                q1 := 427419669081
```

Calcul modulaire :

```
> a2 := a mod 10^12;
> q2 := a2/b mod 10^12;
                                q2 := 427419669081
```

Krandick, Jebelean, 1996 : on gagne encore un facteur 2 en opérant par les deux bouts !

```
> Digits:=6: a1:=evalf(a): q1:=evalf(a1/b);
                                                  12
                                   q1 := .427421 10
> a2 := a mod 10^6: q2 := a2/b mod 10^6;
                                      q2 := 669081
```

Illustration : la division

En naïf :

- représentation classique
- prétraitement du diviseur (Svoboda)

Représentation classique

1. On calcule $c = ab$ ($2k$ mots en base β)

2. On calcule $c \bmod n$ par division euclidienne :

for $j := k-1$ **downto** 0 **do**

$q \leftarrow \lfloor \frac{c_{k+j}\beta + c_{k+j-1}}{n_{k-1}} \rfloor$

$c \leftarrow c - q\beta^j n$

while $c < 0$ **do**

$c \leftarrow c + \beta^j n$

Représentation classique (exemple)

$c = 766970544842443844$, $n = 862664913$, $\beta = 1000$, $k = 3$:

j	c	q	$c - qn\beta^j$	après correction
2	766970544842443844	889	61437185443844	idem
1	061437185443844	071	187976620844	idem
0	187976620844	218	−84330190	778334723

$q = 889071217$, $r = 778334723$.

Représentation classique (suite)

• corrections : Au plus une correction pour chaque i si n est normalisé (i.e. $n_{k-1} \geq \beta/2$) [Knuth vol. 2]

• normalisation : si n n'est pas normalisé, on peut calculer modulo $n' = cn$ normalisé, et ne réduire modulo n que si besoin.

• semi-réduction : Au lieu de réduire complètement $c \bmod n$ à la fin, on peut se contenter de $c < \beta^k$.

Prétraitement du diviseur (Svoboda, 1963)

$$n' = 1160 \cdot n = 1000691299080$$

c	q	$c - qn'\beta^j$	après correction
766970544842443844	766	441009747163844	idem
441009747163844	441	-295115730436	705575568644

$q' = 766440$, $r' = 705575568644$

Dernière itération classique : $r' \bmod n = 778334723$.

Règle 2 : Diviser pour régner

Théorème. Tout algorithme a une variante sous-quadratique.

Preuve. Ajuster la taille du mot de base.

Exemple : la division (Burnikel, Ziegler, 1998).

Pour diviser $2n$ mots par n mots, il suffit de savoir diviser n mots par $n/2$ mots :

RecursiveDivide ($a : 2n$, $b : n$)
$(q_1, r_1) \leftarrow \text{RecursiveDivide}(a_1 : n, b_1 : n/2)$
$a' \leftarrow r_1 \beta^n + a_0 : 3n/2$
$(q_0, r_0) \leftarrow \text{RecursiveDivide}(a'_1 : n, b_1 : n/2)$
$q \leftarrow q_1 \beta^{n/2} + q_0$
$r \leftarrow r_0 \beta^{n/2} + a'_0 : n$
Adjust (q, r) if necessary.

Règle 3 : Ne pas calculer des valeurs connues

Itération de Newton pour le calcul de $\sqrt{a}$:

$$x_{k+1} = x_k + \frac{1}{2x_k}(a - x_k^2).$$

Supposons que x_k a n bits (corrects). On sait donc que les n bits de poids fort de x_k^2 annulent ceux de a : il suffit donc de calculer les n bits de poids faible (produit court).

Mulders, *On short multiplications and divisions*, AAECC, 2000.

Règle 4 : Ne pas calculer des valeurs inutiles

Itération de Newton pour le calcul de $1/a$:

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

Supposons que x_k a n bits corrects. Il faut prendre $2n$ bits de a pour obtenir $2n$ bits corrects dans x_{k+1}. Le produit ax_k fait $3n$ bits : les n bits de poids fort s'annulent avec 1, et les n bits de poids faible sont inutiles.

Hanrot, Quercia, Z., *The Middle Product Algorithm, I*, AAECC, 2004.

Régle 5 : Réutiliser ce qui est connu

Exemple : *The* $x^n - 1$ *trick* (Bernstein, 1999).

Les algorithmes style FFT calculent souvent une convolution cyclique (mod $2^n - 1$) ou négacyclique (mod $2^n + 1$).

On veut calculer un produit $n \times n$ dont une moitié est déjà connue (produit court) : $xy = z_0 + z_1$.

Au lieu de calculer xy sur $2n$ bits, on calcule $xy \bmod 2^n - 1 = z_0 + z_1$. Une simple soustraction donne la moitié cherchée.

Ex. (Schönhage, IPL, 2000) :

$$x_{k+1} = x_k + (x_k - a x_k^2).$$

Le produit $a x_k^2$ fait $4n$ bits, et on le veut à 2^{-2n} près, sachant que les n bits de poids fort donnent x_k. Un calcul modulo $2^{3n} \pm 1$ suffit.

Règle 6 : Attention aux constantes !

Se ramener autant que possible à des multiplications.

Exemple : pour calculer une racine carrée, on peut soit utiliser l'itération directe de Newton :

$$x_{k+1} = x_k + \frac{1}{2}(x_k + \frac{a}{x_k}),$$

ou bien calculer la racine carrée inverse via :

$$x_{k+1} = x_k + \frac{x_k}{2}(1 - a x_k^2).$$

Si la division est coûteuse par rapport à la multiplication, on préfèrera la seconde méthode.

Règle 7 : Se ramener à ce qui est facile à calculer

Si on sait calculer un inverse en $cM(n)$, alors on sait calculer un quotient en $(c+1)M(n)$.

Si on sait calculer une racine carrée inverse en $dM(n)$, alors on sait calculer une racine carrée en $(d+1)M(n)$.

On sait même faire mieux : $(c+3)/2M(n)$ et $(2d+5)/4M(n)$ (meilleur dès que $c > 1$ et $d > 1/2$).

Idée : incorporer le dividende lors de la dernière itération de Newton.

Karp, Markstein, *High-Precision Division and Square Root*, ACM TOMS, 1997.

Division à la Karp-Markstein de a par b sur n bits :

1. $i \leftarrow 1/b$ sur $n/2$ bits $[\frac{c}{2}M(n)]$
2. $q \leftarrow ai$ tronqué à $n/2$ bits $[\frac{1}{2}M(n)]$
3. $q' \leftarrow q + i(a - bq)$ sur n bits $[M(n)]$

Preuve : $q = a/b + \epsilon$, $q = a/b + \epsilon + 1/b(1+\epsilon')(-b\epsilon)(1+\epsilon'')$.

Racine carrée à la Karp-Markstein de a sur n bits :

1. $i \leftarrow a^{-1/2}$ sur $n/2$ bits $[\frac{d}{2}M(n)]$
2. $s \leftarrow ai$ tronqué à $n/2$ bits $[\frac{1}{2}M(n)]$
3. $s' \leftarrow s + \frac{i}{2}(a - s^2)$ $[\frac{3}{4}M(n)]$

Preuve : $s = \sqrt{a} + \epsilon$, $s' = \sqrt{a} + \epsilon + \frac{1}{2}a^{-1/2}(1+\epsilon')(-2s\epsilon - \epsilon^2)$.

Retour sur la division

Algorithme de Barrett (1986).

0. $i = 1/b$ (flottant).

1. $q = \lfloor ai \rceil$ (flottant puis entier).

2. $r \leftarrow a - qb$.

3. Corriger.

Deux produits courts (1 et 2) : $D(n) \leq 2M^*(n)$.

Règle 8 : Changer de domaine au besoin

Exemple : multiplication de Montgomery pour les calculs mod n.

Si $n < \beta^k$, on remplace a par $a' = a\beta^k$.

Additions/soustractions inchangées, multiplications remplacées par REDC.

P. Montgomery, *Modular Multiplication Without Trial Division*, Math. of Comp., 1985.

La multiplication de Montgomery

$a \to a' = \beta^k a \bmod n$

$b \to b' = \beta^k b \bmod n$

$ab \bmod n \to \frac{a'b'}{\beta^k} \bmod n$

$$\mathrm{REDC}(a,b) := \frac{ab}{\beta^k} \bmod n$$

Avantage : $\mathrm{REDC}(a,b)$ se calcule sans division.

On retrouve a par $\mathrm{REDC}(a',1)$.

REDC

On précalcule $n' = -1/n \bmod \beta$.

Input : $c = c_{2k-1}\beta^{2k-1} + \cdots + c_1\beta + c_0$
Output : $\frac{c}{\beta^k} \bmod n$
for $i := 1$ **to** n **do**
  $q \leftarrow n'c_0 \bmod \beta$
  $c \leftarrow \frac{c+qn}{\beta}$
if $c \geq n$ **then**
  $c \leftarrow c - n$

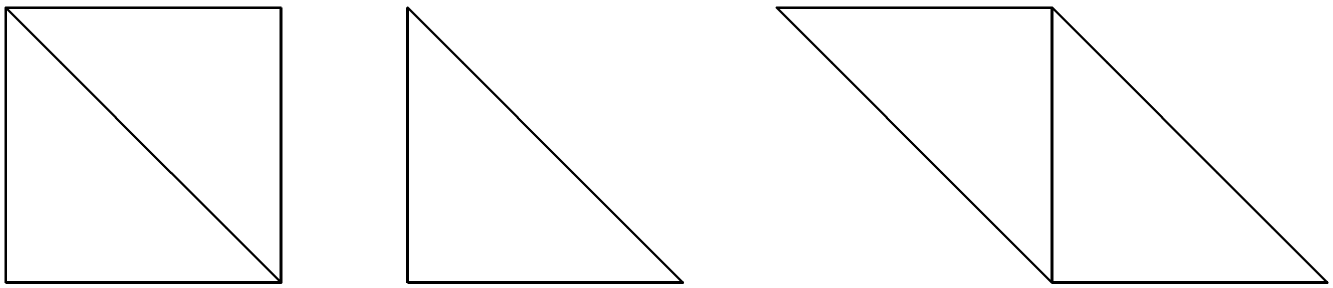
$$c + qn \equiv c_0 + (-c_0/n)n \equiv 0 \bmod \beta$$

Avantages de la multiplication de Montgomery

1. pas de correction dans la boucle (une seule à la fin) ;
2. calcul facile de q par multiplication modulo β ;
3. autres opérations inchangées : $(a+b)' = a' + b'$.

Variante : on peut aussi entrelacer le calcul de $c = ab$ et de $\frac{c}{\beta^k} \bmod n$.

Règle 9 : Faire un dessin

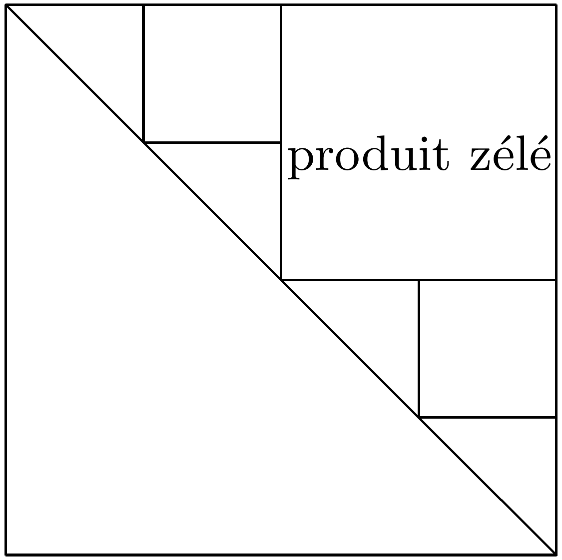


Multiplication   Produit court   Produit médian

Multiplication, produit médian $\leq$ 2 produits courts

Règle 10 : Anticiper

Produit « détendu » :

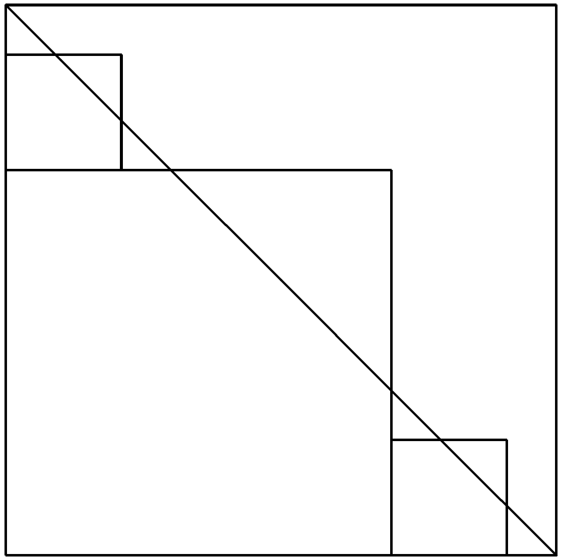


J. van der Hoeven, *Relax, but don't be too lazy*, JSC, 2002.

Règle 11 : Calculer plus au besoin

[exception à la règle 4 : Ne pas calculer des valeurs inutiles]

Exemple : calcul d'un produit court.



Mulders, *On short multiplications and divisions*, AAECC, 2000.

Pour en savoir plus

R. P. Brent, P. Zimmermann, *Modern Computer Arithmetic*, en préparation.

P. Zimmermann, *Arithmétique en précision arbitraire*, Réseaux et Systèmes Répartis, Calculateurs Parallèles, 2001.