Speeding up the Division and Square Root of Power Series

Guillaume Hanrot, Michel Quercia, Paul Zimmermann

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Speeding up the division and square root of power series

Guillaume Hanrot, Michel Quercia, Paul Zimmermann

N° 3973

July 17, 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--3973--FR+ENG

Speeding up the division and square root of power series

Guillaume Hanrot*, Michel Quercia†, Paul Zimmermann*

Thème 2 — Génie logiciel
et calcul symbolique
Projet PolKA

Rapport de recherche n° 3973 — July 17, 2000 — 20 pages

Abstract: We present new algorithms for the inverse, quotient, or square root of power series. The key trick is a new algorithm — `RecursiveMiddleProduct` or `RMP` — computing the n middle coefficients of a $2n \times n$ product in essentially the same number of operations — $K(n)$ — than a full $n \times n$ product with Karatsuba's method. This improves previous work of Mulders, Karp and Markstein, Burnikel and Ziegler. These results apply both to series, polynomials, and multiple precision floating-point numbers.

Key-words: division, square root, Newton method, Karatsuba's algorithm

* e-mail: {hanrot,zimmerma}@loria.fr
† 23 rue de Montchapet, 21000 Dijon, e-mail: quercia@cal.enst.fr

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Nouveaux algorithmes de division et racine carrée de séries formelles

Résumé : Nous présentons de nouveaux algorithmes pour l'inverse, le quotient ou la racine carrée de séries de Taylor. Tous reposent sur un nouvel algorithme — baptisé `RecursiveMiddleProduct` ou `RMP` — calculant les n termes médians d'un produit $2n \times n$ en pratiquement le même nombre d'opérations qu'un produit complet $n \times n$ avec la méthode de Karatsuba. Cela améliore des travaux de Mulders, Karp et Markstein, Burnikel et Ziegler. Ces résultats s'appliquent aussi bien aux séries de Taylor qu'aux polynômes et aux nombres flottants en précision arbitraire.

Mots-clés : division, racine carrée, méthode de Newton, algorithme de Karatsuba

Introduction

One of the major tools for performing arithmetic or computing special functions is Newton's iteration. Indeed its quadratic rate of convergence (so that the number of correct digits or terms doubles at each step) and its self-correcting character make it especially suitable for this kind of computations.

In full generality, Newton's rule for computing a root of a differentiable function $f(x)$ can be written as $x_{k+1} = x_k - f(x_k)/f'(x_k)$. As a rule of thumb, the term x_k should be seen as the main term, whereas $f(x_k)/f'(x_k)$ should be seen as the correcting term. In particular, not all the digits of $f(x_k)/f'(x_k)$ are significant, since the low weight ones will be corrected in the next iteration. Furthermore, since $f(x_k)$ is supposed to tend to zero quickly, one can expect important cancellations in the evaluation of $f(x_k)/f'(x_k)$. The present paper shows how to evaluate only the meaningful part of $f(x_k)/f'(x_k)$, i.e., the "middle" digits, when the underlying arithmetic is mainly based on Karatsuba multiplication. As a consequence, new algorithms for inversion, division and square root are derived, using Newton's iteration.

The paper is organized as follows. Section 1 describes the basic trick, proves its correctness and analyzes its complexity. Section 2 shows how this trick can be used to compute the inverse of a power series of order n in $\sim 0.904K(n)$ ring operations. In Section 3, it is shown that this trick may also be used directly for division and square root, giving algorithms of complexity $\sim K(n)$ and $\sim 0.789K(n)$ respectively. Previous best-known algorithms for Karatsuba multiplication were Mulder's short division algorithm [4] with a complexity of $\sim 1.397K(n)$, and Zimmermann's square root algorithm with a complexity of $K(n)$ [7]. In Section 4, we estimate the exact number of ring multiplications occuring in a call to our method when n is not a power of 2. Finally, Section 5 discusses the case of floating-point numbers.

1 Newton iteration for the inverse and the basic trick

Newton's iteration for computing $1/A$ — where A is a number, a polynomial, or a series — follows the recurrence:

$$x_{k+1} = x_k + x_k(1 - Ax_k). \tag{1}$$

Suppose we are looking for an approximation of $1/A$ to precision n. In the last step of Newton's iteration, x_k is then accurate to precision $n/2$, and we have to compute Ax_k to precision n, where the $n/2$ most significant coefficients — or bits — of Ax_k vanish with 1. To get an approximation x_{k+1} to precision n of $1/A$, we multiply x_k by the $n/2$ most significant coefficients of $1 - Ax_k$. The cost of this last step was $3M(n/2)$ so far: $2M(n/2)$ for the product Ax_k which splits into two products of $n/2$ coefficients, and $M(n/2)$ for the product of x_k by $1 - Ax_k$. The total cost of Newton's iteration to compute $1/A$ to precision n is thus $3M(n)$ for FFT multiplication and $\frac{3}{2}K(n)$ for Karatsuba multiplication [2].

As we know in advance that the upper $n/2$ bits of Ax_k will vanish with those of 1, a natural question is the following: is there a faster way to compute directly the $n/2$ required

bits — from position $n/2$ to n — of the product Ax_k? We give here a positive answer under the Karatsuba model, i.e. $M(n) \sim 3M(n/2)$.

Basic trick. For the sake of clarity, we suppose A is a Taylor series in the variable t (at $t = 0$). Assume we have $A = a_0 + a_1 t + a_2 t^2 + a_3 t^3$ and $x = x_0 + x_1 t$. We have

$$\begin{aligned} Ax = a_0 x_0 \quad + \quad & (a_0 x_1 + a_1 x_0) t + (a_1 x_1 + a_2 x_0) t^2 + \\ & (a_2 x_1 + a_3 x_0) t^3 + a_3 x_1 t^4 \end{aligned}$$

and we want only

$$(a_0 x_1 + a_1 x_0) t + (a_1 x_1 + a_2 x_0) t^2.$$

The algorithm works as follows (see Fig. 1):

Algorithm `MiddleProduct`.
Input: $A = a_0 + a_1 t + a_2 t^2 + a_3 t^3, x = x_0 + x_1 t$
Output: $h = a_0 x_1 + a_1 x_0$ and $l = a_1 x_1 + a_2 x_0$
1. Compute $\alpha = (a_0 + a_1) x_1$
2. Compute $\beta = a_1 (x_0 - x_1)$
3. Compute $\gamma = (a_1 + a_2) x_0$
4. $h = \alpha + \beta$
5. $l = \gamma - \beta$.

Now we can use algorithm `MiddleProduct` recursively (note that here and in the sequel we use `RMP` as a shortcut for `RecursiveMiddleProduct`):

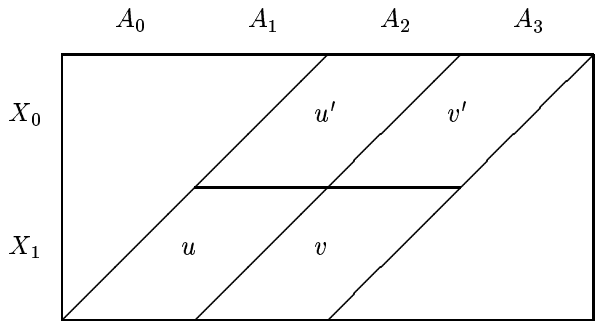
Algorithm `RMP`$([a_0, \ldots, a_{2n-1}], [x_0, \ldots, x_{n-1}])$
0. If $n = 1$, return $[a_0 x_0]$
1. $\alpha =$ `RMP`$([a_0 + a_{n/2}, \ldots, a_{n-1} + a_{3n/2-1}], [x_{n/2}, \ldots, x_{n-1}])$
2. $\beta =$ `RMP`$([a_{n/2}, \ldots, a_{3n/2-1}], [x_0 - x_{n/2}, \ldots, x_{n/2-1} - x_{n-1}])$
3. $\gamma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

Theorem 1 *Algorithm* `RMP` *returns* $[c_{n-1}, \ldots, c_{2n-2}]$ *where* $c_k = \sum_{i+j=k} a_i x_j$, *using exactly* $K(n) = n^{\log_2 3}$ *ring multiplications when* n *is a power of two.*

1.1 Proof of the theorem

In this section, we give a formal description of the trick presented above, and prove that it gives the correct result for polynomials and power series. See also Fig. 1.

Let $A = \sum_{k=0}^{2n-1} a_k t^k$ and $x = \sum_{k=0}^{n-1} x_k t^k$ be polynomials with coefficients in any base ring.



$$\begin{array}{lcll} u+v & = & \alpha & = & \texttt{RMP}([A_0,A_1]+[A_1,A_2],\ X_1) \\ u'-v & = & \beta & = & \texttt{RMP}([A_1,A_2], \quad X_0-X_1) \\ u'+v' & = & \gamma & = & \texttt{RMP}([A_1,A_2]+[A_2,A_3],\ X_0) \end{array}$$

Figure 1: `MiddleProduct` recursion

One has

$$Ax = \sum_{k=0}^{3n-2} \left(\sum_{l=\max(0,k-n+1)}^{\min(k,2n-1)} a_l x_{k-l} \right) t^k.$$

Our method has two variants, according to the choice of a parameter $\delta \in \{-1, 0\}$, corresponding to the computation of the terms of the product whose degree is in the interval $[n+\delta, 2n+\delta-1]$. The case presented in the introduction and algorithm `RecursiveMiddleProduct` corresponds to $\delta = -1$.

Define a pseudo-product $*_\delta$ by the formula

$$\left(\sum_{k=0}^{2n-1} a_k t^k \right) *_\delta \left(\sum_{j=0}^{n-1} x_j t^j \right) = \sum_{k=n+\delta}^{2n+\delta-1} \left(\sum_{l=k-n+1}^{k} a_l x_{k-l} \right) t^k.$$

One readily checks that this quantity corresponds to n middle terms of the usual product Ax.

Assume in the sequel that n is even. Put

$$\begin{array}{rcl} A^{(h)} & = & \displaystyle\sum_{0 \le k \le n-1} a_k t^k, \\ A^{(m)} & = & \displaystyle t^{-n/2} \sum_{n/2 \le k \le n+n/2-1} a_k t^k, \\ A^{(l)} & = & \displaystyle t^{-n} \sum_{n \le k \le 2n-1} a_k t^k, \end{array}$$

$$\begin{aligned} x^{(h)} &= \sum_{0\le k\le n/2-1} x_k t^k, \\ x^{(l)} &= t^{-n/2} \sum_{n/2\le k\le 2n/2-1} x_k t^k. \end{aligned}$$

Our theorem then follows from the following identity:

$$\begin{aligned} A *_\delta x &= \left((A^{(h)} + A^{(m)}) *_\delta x^{(l)} + A^{(m)} *_\delta (x^{(h)} - x^{(l)})\right) t^{n/2} \\ &+ \left((A^{(m)} + A^{(l)}) *_\delta x^{(h)} - A^{(m)} *_\delta (x^{(h)} - x^{(l)})\right) t^n. \end{aligned}$$

This identity can be rewritten, using the bilinearity of $*$:

$$\begin{aligned} A *_\delta x &= \left(A^{(h)} *_\delta x^{(l)} + A^{(m)} *_\delta x^{(h)}\right) t^{n/2} \\ &+ \left(A^{(l)} *_\delta x^{(h)} + A^{(m)} *_\delta x^{(l)}\right) t^n. \end{aligned}$$

We now evaluate separately the two parts of the right hand side. The quantity $(A^{(h)} *_\delta x^{(l)} + A^{(m)} *_\delta x^{(h)})$ evaluates to

$$\sum_{k=n/2+\delta}^{n-1+\delta} \sum_{l=k-n/2+1}^{k} (a_l x_{n/2+k-l} + a_{n/2+l} x_{k-l}) t^k,$$

which is

$$\sum_{n+\delta\le k\le 3n/2-1+\delta} t^{k-n/2} \left(\sum_{l=k-n+1}^{k-n/2} a_l x_{k-l} + \sum_{l=k-n+1}^{k-n/2} a_{n/2+l} x_{k-l-n/2} \right),$$

or, finally,

$$t^{-n/2} \sum_{n+\delta\le k\le 3n/2-1+\delta} t^k \left(\sum_{l=k-n+1}^{k} a_l x_{k-l} \right).$$

The second term is handled the same way, and becomes

$$t^{-n} \sum_{3n/2+\delta\le k\le 2n-1+\delta} t^k \left(\sum_{l=k-n+1}^{k} a_l x_{k-l} \right),$$

which concludes the first part of the proof of Theorem 1. The number $K(n)$ of ring multiplications satisfies $K(1) = 1$ and $K(2^m) = 3 \cdot K(2^{m-1})$, i.e., $K(n) = n^{\log_2 3}$ as claimed, for n being a power of two. ■

Remark. – Theorem 1 assumes n is a power of two. Algorithm `RecursiveMiddleProduct` works for odd sizes too, but then the number of ring multiplications can be slightly larger than $K(n)$, see Section 4. Except in this last section, we therefore assume in the rest of the paper that n is a power of two for the complexity results.

2 Application to inverse power series computation

Theorem 2 *Using algorithm* `RecursiveMiddleProduct`*, one can compute the inverse of a power series to order n in $\frac{1+a}{2}K(n)$ operations, where $a \cdot K(n)$ is the number of operations needed to multiply two power series of order n. Using a full product ($a = 1$), an inverse needs $K(n)$ operations; using Mulders' short product algorithm [4], which achieves $a \sim 0.808$, an inverse needs only $\sim 0.904K(n)$ operations.*

Proof. The proof is the following MuPAD procedure: `A` is the list of coefficients from the input power series, where `A[i+1]` is the coefficient of degree i, `x` contains the successive approximations of $1/A$ using Newton's iteration, k is the number of coefficients in x, `RMP(A, 2, x, 1, k)` calls the `RecursiveMiddleProduct` algorithm on A and x, and `karamul(x, y, k)` multiplies x by y using Karatsuba's algorithm.

```
karainv := proc(A) local n, k, y, i, x;
begin
   n:=nops(A);
   x:=[1/A[1]];
   k:=1;
   while k<n do
      y:=RecursiveMiddleProduct(A, 2, x, 1, k);
      y:=karamul(x, y, k);
      x:=append(x, -y[i] $ i=1..k);
      k:=2*k;
   end_while;
   [x[i] $ i=1..n]
end_proc:
```

The total cost is therefore $2K(n/2)$ for the last step, and $2K(n/2)\cdot(1+1/3+1/9+\ldots) = K(n)$ for the whole inversion.

For the `karamul(x, y, k)` call, x and y have both k elements, but we need only the k most significant terms of the product. Thom Mulders [4] designed an algorithm to compute those k most significant terms in $a \cdot K(k)$ operations together with Karatsuba multiplication, where $a = \frac{\beta^\alpha}{1-2(1-\beta)^\alpha}$ with $\beta = 1 - (1/2)^{(1/(\alpha-1))}$ and $\alpha = \frac{\log 3}{\log 2}$, giving $a \sim 0.808$. If we use that algorithm instead of a full product, then the inversion cost becomes $(1+a)/2 \sim 0.904$. ■

Corollary 1 *The quotient of two degree n polynomials or two order n power series — called short division by Mulders — can be computed in $\frac{3+5a}{6}K(n)$ operations, if a short product can be done in $a \cdot K(n)$ operations. With $a \sim 0.808$ from Mulders, this gives $\sim 1.173K(n)$.*

PROOF. We use Karp and Markstein's trick [3] to incorporate the dividend in the last Newton iteration. This can be done in turn using a short multiplication. Thus the total cost is equivalent to that of one inversion plus one short multiplication of size $n/2$, i.e. $(1+a)/2K(n) + aK(n/2) = (3+5a)/6K(n) \sim 1.173K(n)$. ∎

This improves on the previous best-known algorithm in $\sim 1.397K(n)$ from Mulders [4, Table 6].

Remark. – If k quotients with the same divisor have to be computed, this method costs $\frac{1+a+2k+4ka}{6}K(n)$, whereas the alternative methods which consists of first computing the divisor inverse up to full precision, and then using Mulders' short product algorithm between each numerator and that precomputed inverse costs $\frac{1+a+2ka}{2}K(n)$. The latter method is faster as soon as $k > \frac{1+a}{1-a}$, i.e. $k \geq 10$ for $a \sim 0.808$.

Corollary 2 *The square root of a power series of order n can be computed in $\frac{12+5a}{18}K(n)$ operations, if a short product can be done in $a \cdot K(n)$ operations. With $a \sim 0.808$ from Mulders, this gives $\sim 0.891K(n)$.*

PROOF. We use the algorithm `SqrtRem` described in [7]. It needs $\frac{3}{2}K(n/2)$ to get an approximation to order $n/2$ of the square root, with remainder. Then as shown in [7], a single short division of size $n/2$ suffices to get an approximation to order n. The total cost is thus $\frac{3}{2}K(n/2) + \frac{3+5a}{6}K(n/2) \sim \frac{12+5a}{18}K(n)$. ∎

Remark. – using the iteration $x_{k+1} = x_k + \frac{A-x_k^2}{2x_k}$ with short products and divisions gives $\frac{3+11a}{12}K(n) \sim 0.990K(n)$. Using the inverse square root iteration $x_{k+1} = x_k + \frac{x_k}{2}(1 - Ax_k^2)$ with Karp/Markstein idea gives $\frac{1+4a}{3}K(n) \sim 1.410K(n)$ (the last step on its own with the computation of $y_k = Ax_k$ requires three short products of size $n/2$, i.e. $a \cdot K(n)$ operations).

Remark. – in the above analysis, a square was considered to be as expensive as a multiplication. A better complexity might be obtained if one considers a different model.

Although the number of multiplications used by `RMP` and Karatsuba's full product are equal, the former is slightly more efficient: Algorithm `RMP` on inputs of size n and $2n$ uses only $6n$ additions/subtractions in the main call, whereas a careful implementation of Karatsuba's algorithm for inputs of size n needs $7n$ additions/subtractions.

3 Direct division and square-root using Algorithm `RMP`

The Algorithm `RMP` may also be used to directly — i.e. without Newton's iteration — compute the quotient or square root of series or polynomials. These new algorithms are described and analyzed in this section.

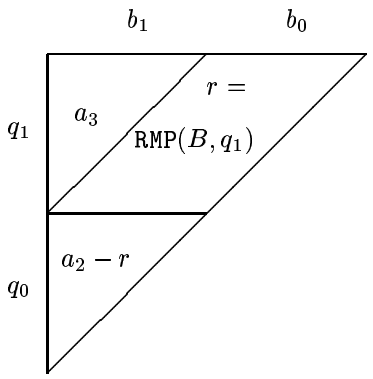


Figure 2: Algorithm `Divide`.

Algorithm `Divide`.
Input: $A = a_3x^{3n} + a_2x^{2n} + a_1x^n + a_0$ and $B = b_1x^n + b_0$ with $\deg(a_i) < n, \deg(b_1) = n, \deg(b_0) < n$
Output: Q such that $\deg(A - Q \cdot B) < 2n$.
0. If $n = 1$ then return $\mathrm{quo}(A, B)$
1. $q_1 \leftarrow$ `Divide`$(a_3x^n + a_2, b_1)$
2. $r \leftarrow$ `RMP`(B, q_1)
3. $q_0 \leftarrow$ `Divide`$(a_2 - r, b_1)$
Return $Q = q_1x^n + q_0$.

Theorem 3 *Algorithm* `Divide` *with input A of degree $< 2n$ and B of degree n correctly computes their quotient, in $\sim K(n)$ arithmetic operations.*

Proof. After step 1, we have by induction $\deg(a_3x^n + a_2 - q_1b_1) < n$, therefore $\deg(A - q_1x^nB) < 3n$. This implies q_1B has the form $a_3x^{2n} + rx^n + s$. Algorithm `RMP` on inputs B and q_1 in step 2 precisely determines the middle part r (see Fig. 2). We then have $A - q_1x^nB = (a_2 - r)x^{2n} +$ low order terms, and the second division in step 3 finds the low significant part of the quotient.

Let $D(n)$ be the cost of `Divide` with a dividend of degree $2n$ and a divisor of degree n. The call to `RMP`(B, q_1) needs $K(n/2)$ arithmetic operations since q_1 has degree $< n/2$. We have $D(n) \leq K(n/2) + 2D(n/2) \sim K(n/2)(1 + \frac{2}{3} + \frac{4}{9} + \cdots) \sim K(n)$. ■

Remark. – The algorithm from Corollary 1 would be faster if we knew how to compute a short product in less than $\frac{3}{5}K(n)$ operations; however the latter seems very unlikely.

Corollary 3 *Using algorithm* `Divide` *and Mulders' algorithm, a division with remainder of a polynomial of degree $< 2n$ by a polynomial of degree n is computed in $(1 + a)K(n)$ arithmetic operations.*

The same idea works for square-root too:

Algorithm `SquareRoot`.
Input: $A = a_5x^{5n} + a_4x^{4n} + a_3x^{3n}$ with $\deg(a_i) < n$, $\deg(A)$ even and $\mathrm{lt}(A)$ is a square.
Output: $R = r_2x^{2n} + r_1x^n + r_0$ such that $\deg(A - R^2) < 3n$.
1. $(r_2, A') \leftarrow$ `SqrtRem`$(a_5x^n + a_4)$
2. $r_1x^n + r_0 \leftarrow$ `Divide`$(A'x^{3n} + a_3x^{2n}, 2r_2x^n + r_1)$
3. Return $R = r_2x^{2n} + r_1x^n + r_0$.

PROOF. After Step 1, we have $a_5x^n + a_4 = r_2^2 + A'$, thus $A = (r_2x^{2n})^2 + A'x^{4n} + a_3x^{3n}$. It follows from Step 2 that $A'x^{3n} + a_3x^{2n} = (r_1x^n + r_0)(2r_2x^n + r_1) + r'$ with $\deg(r') < 2n$. Hence $A - R^2 = r'x^n - r_0^2 - r_1r_0x^n$ whose degree is less than $3n$.

In step 1, one calls algorithm `SqrtRem` from [7] with an input of degree $< 2n$ and outputs r_2 and A' of degree $< n$, the cost of that step is $\frac{3}{2}K(n)$. In step 2, r_1 is used both in the input and output, but its coefficients are computed before they are needed; the cost of this step is $K(2n)$. The total cost is $S(3n) = (\frac{3}{2}(1/3)^{\log_2 3} + (2/3)^{\log_2 3})K(3n) \sim 0.789K(3n)$. ■

Remark. – The algorithm from Corollary 2 would be faster if we knew how to compute a short product in less than $\sim 0.44K(n)$ operations; however the latter is not possible, otherwise, using two short products for the high and low parts, we could compute a full-product in less than $K(n)$ operations!

4 Precise count of number of ring operations

In this section, we analyze more carefully the number of ring mulplications or divisions used by the new algorithms for division and square root, with respect to Karatsuba's multiplication algorithm. The latter satisfies the following recurrence for its number of multiplications:

$$\begin{aligned} K(2n) &= 3K(n), \\ K(2n+1) &= 2K(n+1) + K(n). \end{aligned}$$

In fact, a similar recurrence holds for the number of operations $R(n)$ of Algorithm `RMP`:

$$\begin{aligned} R(2n) &= 3R(n), \\ R(2n+1) &= \min(R(2n+2), R(2n) + 4n + 1, 2R(n+1) + \min(R(n) + n, R'(n))). \end{aligned}$$

where R' is defined by

$$\begin{aligned} R'(1) &= 2, \\ R'(2n) &= 3R'(n), \\ R'(2n+1) &= R'(n) + 2R(n+1). \end{aligned}$$

For odd sizes the first term $R(2n+2)$ corresponds to calling `RMP` with size $2n+2$, by adding zeroes to the two input lists; the second term $R(2n) + 4n + 1$ consists in calling `RMP`

with size $2n$, and computing separately the $4n+1$ remaining products (cf the implementation in Section 6). The third term comes from the following way to compute a middle product for odd size: two calls to `RMP` with size $n+1$, one with size n, and a separate computation of the n missing products, with a total of $2R(n+1)+R(n)+n$ ring multiplications. This saves one operation for $n=11$, with 71 multiplications instead of 72, but the average gain of that third way is rather small. The fourth term for $R(2n+1)$ consists in calling twice `RMP` with size $n+1$, and a modified version, say `RMP'`, with size n. `RMP'` takes arguments of length n and $2n$ respectively, where `RMP` takes arguments of length n and $2n-1$.

For our division algorithm, we have the following recurrence:

$$\begin{aligned} D(2n) &= 2D(n)+R(n) \\ D(2n+1) &= \min(D(2n+2), D(n+1)+\min(R(n)+n, R(n+1))+D(n)). \end{aligned}$$

Indeed, for even size $2n$, one first calls the algorithm recursively on size n, with cost $D(n)$, then one calls `RMP` on the $2n$ divisor coefficients and the n most significant coefficients from the quotient just computed (cost $R(n)$), and finally one calls `Divide` with cost $D(n)$ to get the lower n coefficients from the quotient. For odd size $2n+1$, either one calls `Divide` with size $2n+2$ after adding random coefficients to the inputs, with cost $D(2n+2)$; or one first calls `Divide` to get the $n+1$ most significant coefficients from the quotient, then one call to `RMP` with size n with n auxiliary products — or one call to `RMP` with size $n+1$ — is needed to compute the corresponding terms to subtract from the dividend before the final short division of size n. See Section 6 for more details about the implementation.

Theorem 4 *For all $n \geq 1$, we have $D(n) \leq R(n) \leq 1.21653K(n)$.*

Proof. From the definition of R and the obvious fact $R'(n) \geq R(n)$, it easily follows that

$$\begin{aligned} R(2n+1)-R(2n) &\leq 4n+1, \\ R(2n+2)-R(2n+1) &\leq \max(0, R(n+1)-R(n), 3(R(n+1)-R(n))-4n-1). \end{aligned}$$

From this, we easily deduce by induction that $R(n+1)-R(n) \leq 2n+1$.

Let us prove by induction the first inequality of the theorem. One has $D(1)=1=R(1)$. Assume now that $D(k) \leq R(k)$ for all $k<n$.

If n is even,

$$D(n) = 2D(n/2)+R(n/2) \leq 3R(n/2) \leq R(n).$$

If n is odd, let us write $n=2p+1$.

- on the one hand $D(2p+1) \leq D(2p+2) \leq R(2p+2)$,
- on the other hand $D(2p+1) \leq D(p+1)+R(p+1)+D(p) \leq 2R(p+1)+R(p) \leq 2R(p+1)+R'(p)$,
- finally $D(2p+1) \leq D(p+1)+R(p)+D(p)+p \leq 3R(p)+3p+1 \leq R(2p)+3p+1$.

From all this $D(2p+1) \leq \min(R(2p+2), 2R(p+1)+R(p), R(2p)+3p+1) \leq R(2p+1)$ follows for n odd, and hence by induction for all n.

As for the rightmost inequality of the theorem, an exhaustive computation of $R(n)/K(n)$ for $n < 2^{26}$ (which takes a few seconds on a Pentium II 400 with 2 GB RAM) shows that in this range $R(n)/K(n) \leq 1.2164494$ (this value being obtained for $n = 43257921$, $R(n) = 2114121307217$, $K(n) = 1737944316251$).

Put now $\theta_l = \max_{x \in [2^l, 2^{l+1}-1[} R(x)/K(x)$. We have $\theta_{25} \leq 1.2164494$.

Now, $R(2n)/K(2n) = R(n)/K(n)$ for any n in $[2^l, 2^{l+1}-1[$. Furthermore,

$$\frac{R(2n+1)}{K(2n+1)} \leq \frac{2R(n+1)+R(n)}{2K(n+1)+K(n)} + \frac{n}{K(2n+1)}. \tag{2}$$

The first term is at most θ_l; since K is increasing, the second term can be seen to be at most $2^{l+1}/K(2^{l+1})$, i.e., $(2/3)^{l+1}$. Hence we have

$$\theta_{l+1} \leq \theta_l + (2/3)^{l+1} \leq \theta_{25} + \sum_{j \geq 26} (2/3)^j \leq 1.21653,$$

which concludes the proof. ■

Remark. – Note that heuristic computations seem to indicate that

$$\limsup R(n)/K(n) = 1.216449773714734305 57...$$

which does not correspond to any constant known to us. Similar computations seem to show that $\lim D(n)/R(n) = 1$. Note that $\liminf R(n)/K(n) = 1$.

Remark. – Computations for $n < 2^{26}$ indicate that the average $n^{-1} \sum_{k=1}^{n} R(k)/K(k)$ seems to tend to a limit close to 1.08021.... M. Quercia has written a detailed study of the comparative behavior of practical implementations of the various algorithms and variants, see [5].

Note on Mulders' short product algorithm. While Mulders' short product algorithm has a theoretical complexity of $aK(n)$ with $a \sim 0.808$, the reality is much more complex, and the speedup heavily depends on the size n. Let us first recall Mulders' algorithm:

Algorithm ShortProduct.
Input: two series s and t of order n (i.e. coefficients from degree 0 to $n-1$)
Output: their product up to order n
if $n \leq 2$ **then** use a full Karatsuba product
else
  let $n/2 \leq k < n$
  let s_0 and t_0 be the most significant parts of s and t up to order k
  compute the full product of s_0 and t_0 using Karatsuba's algorithm
  call ShortProduct to compute the $n-k$ most significant terms of $s(t-t_0)$
    and $(s-s_0)t$
  add the three contributions

Let $S(n)$ be the number of coefficients multiplications from Mulders' algorithm when the theoretically optimal value of k is used — i.e. $k = \lfloor \beta_{\text{opt}} n \rceil$ with $\beta_{\text{opt}} \sim 0.6942363715$ —, and $S_{\text{opt}}(n)$ be value corresponding to the optimal value of k. The first values are:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$K(n)$	1	3	7	9	17	21	25	27	43	51	59	63	71	75	79	81
$RMP(n)$	1	3	8	9	18	24	26	27	44	54	70	72	76	78	80	81
$D(n)$	1	3	6	9	14	20	24	27	36	46	57	64	70	74	78	81
$S(n)$	1	3	5	9	13	15	23	27	31	35	37	45	61	69	77	85
$S_{\text{opt}}(n)$	1	3	5	9	11	15	19	27	29	33	37	45	53	57	73	81

It appears that for $n = 16$, Mulders' algorithm with the theoretically optimal k is less efficient than a full product. We conjecture than for n being a power of two, no speedup can be obtained at all. Inversely, for n near the geometric mean of two consecutive powers of two, like $n = 3, 6, 11$, a good speedup is obtained; $K(43) = 523$ whereas $S_{\text{opt}}(43) = 317$, giving a speedup of almost 40%! For $512 \leq n < 1024$, the ratio $S(n)/K(n)$ averages ~ 0.832, whereas the ratio $S_{\text{opt}}/K(n)$ averages ~ 0.751. Figure 4 clearly shows that the values of n chosen by Mulders only give by accident the expected average gain in multiplications of 20% (in fact the speedup is only about 15% for those values for $S(n)$, and about 30% for S_{opt}). If Mulders would have chosen $n = 32, 64, 128, 256, 512, 1024$ instead of $n = 50, 100, 200, 400, 800, 1600$, he would have obtained instead a increase of the number of multiplications of about 2% to 4%. Note however that the implementations described in [5] actually showed a 20% gain in terms of CPU time, for any n between 500 and 1000, as claimed by Mulders.

5 Floating-point numbers

Unfortunately, the present method does not apply directly to floating-point numbers, where "coefficients" represent machine words — usually 32-bit or 64-bit — from a multiple-precision mantissa; we shall study in this section a modification which however allows one to apply the method in this context; the gain over a full $n \times 2n$ product, though less important than in the polynomial case, remains significant.

One could think that the problem comes from the contribution (via carries) of the products neglected on the lower side. This is however not really a problem: only $O(\log n)$ bits are wrong, so that we can get a correct result by applying our method with n replaced by $n + O(\log n)$, without change on the main term of the asymptotic complexity.

However, the problem definitely comes from carries, and is best illustrated on a simple example. Let M be 2^w, where w is the length of a machine word. Assume that one wishes to perform `RMP` on $aM^3 + 2^{w-1}M^2 + 2^{w-1}M + b$ and $a'M + b'$. Part of the computation amounts then to call `RMP` on $(a + 2^{w-1})M + 2^w = (a + 2^{w-1} + 1)M$ and b', which will yield $(a + 2^{w-1} + 1)b'$. However the correct answer should be $(a + 2^{w-1})b'$.

More generally, the trouble comes from the carries in the definition of α, β and γ: the carry from word m is propagated to the word $m + 1$, and is used in the wrong `RMP` call when

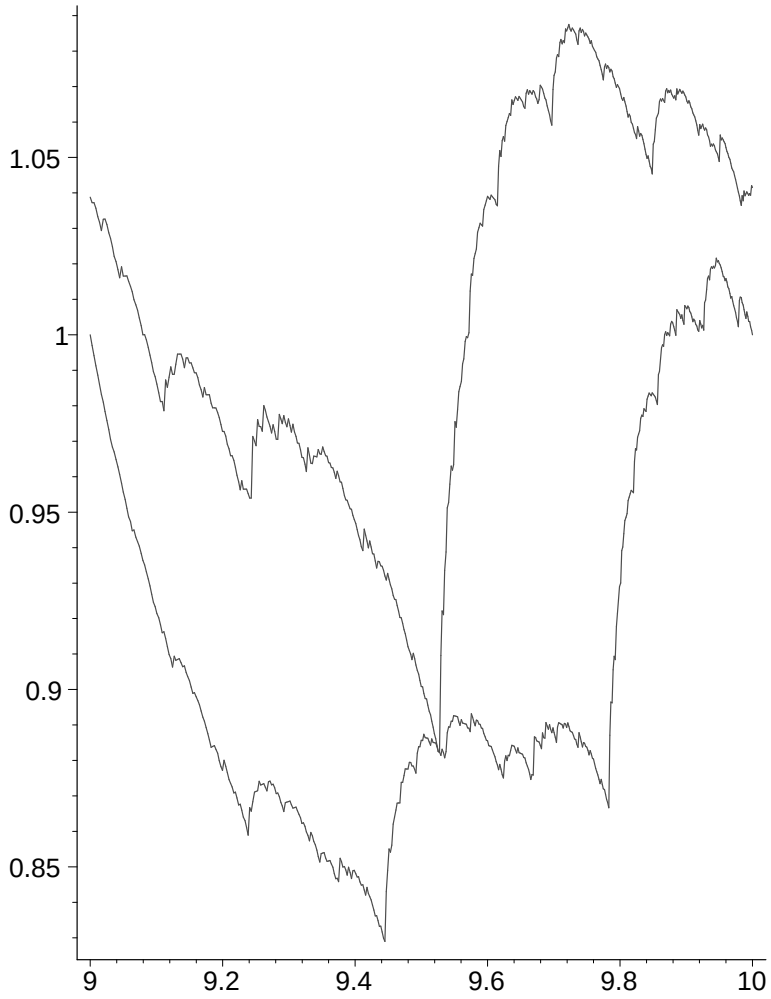


Figure 3: Plot of the ratio $S(n)/n^{\log_2 3}$ and $S_{\text{opt}}/n^{\log_2 3}$ with respect to $\log n$ for $n = 512$ to 1024.

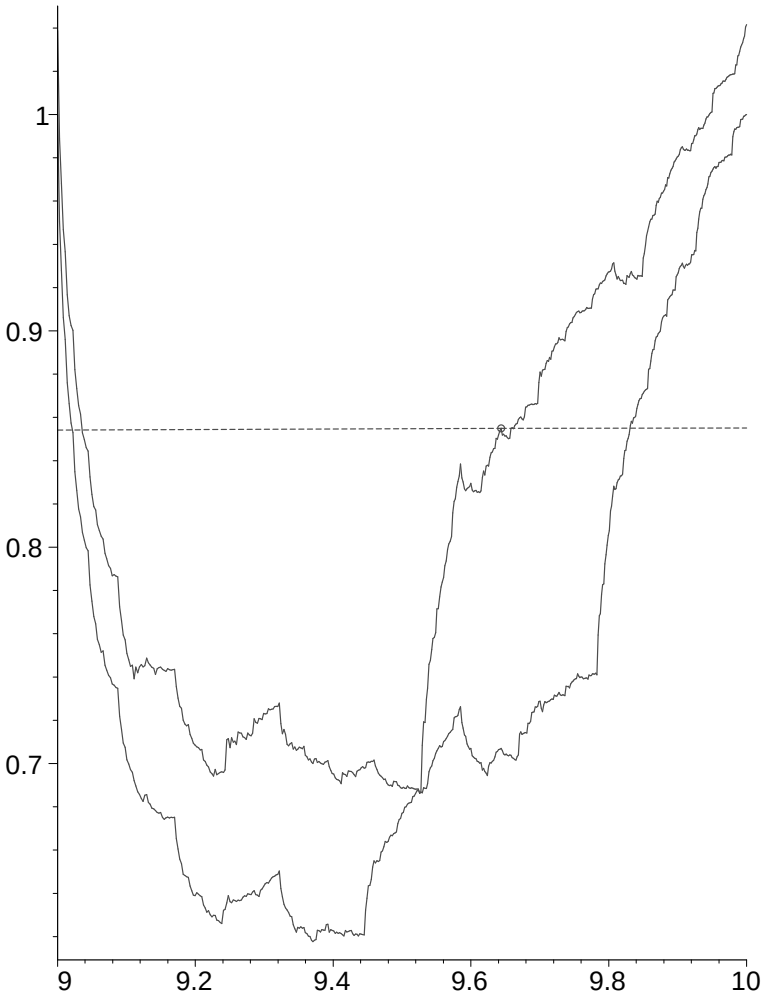


Figure 4: Plot of the ratio $S(n)/K(n)$ and $S_{\text{opt}}/K(n)$ with respect to $\log n$ for $n = 512$ to 1024. The dash line represents the values given by Mulders, the circle corresponds to $n = 800$.

(at a deeper level of recursion) the words m and $m+1$ are separated. Synthetically, we can say that *these carries break the bilinearity*. On the other hand, carries coming from the RMP call itself and occuring after the RMP call can be propagated without any incidence on the results, at least when the odd case is treated by method 1 ($R(2n+1) = 3R(n) + 4n + 1$).

The same problem appears for p-adic numbers (though the carries are propagated the reverse way).

A workaround is the following: divide the input numbers of Algorithm RMP into 2^k chunks of m words and consider them as coefficients of two polynomials of degree $2^k - 1$. Apply Algorithm RMP for polynomials with integer arithmetic on the coefficients. This will yield at the bottom level of recursion signed coefficients of at most m words plus k bits due to the carries so if $k+1$ does not exceed the bit-size of a word, it is enough to enlarge each chunk with an $(m+1)$-st word filled with zeroes at the beginning of Algorithm RMP and to do 2-complement arithmetic on each enlarged chunk independently, that is to say without propagating carries (caused by sign changes) from one chunk to the adjacent ones. This method is basically a carry-save technique.

In practice the optimal value of m, i.e. the point where one stops the recursion in Algorithm RMP, will be near from the cut-off point between Karatsuba's and naive multiplication, since RMP is very similar to Karatsuba's algorithm.

6 Conclusion

This paper presents a new algorithm to compute a short division of two degree n polynomials or series in at most the same number $K(n)$ of arithmetic operations as a full product using Karatsuba's algorithm. The previous best known result was Mulders' algorithm, with a complexity of $\sim 1.397K(n)$.

In addition a new square root algorithm without remainder is presented, with an asymptotic complexity of $\sim 0.789K(n)$. Both algorithms use a new algorithm to compute the n middle coefficients of a $2n \times n$ product. A detailed analysis of the number of coefficient operations used by those algorithms with respect to previously known algorithms is given.

Our algorithms need only a $O(n)$ memory space, i.e. proportional to the input size, in the polynomial/power series version. If one enables a larger memory usage, then faster algorithms exist. In particular, Joris van der Hoeven showed that a division with remainder can be performed in exactly $K(n)$ operations with so-called *relaxed* algorithms [6].

Note that for floating-point numbers $O(n \log n)$ memory space is needed, due to the need of $O(\log n)$ bits per word to store the carries.

Historical note. Algorithm RMP was originally designed by the first and last authors, who obtained algorithms in $\sim 1.173K(n)$ and $\sim 0.891K(n)$ for division and square-root. The second author invented the direct division and square-root algorithms using algorithm RMP, which both gave better complexities. It was then decided to put all results together in a common paper.

Appendix: MuPAD source code of the algorithms

We implemented all the algorithms mentioned in this paper in the MuPAD computer algebra system [1]. We used lists to represent power series, with $l[i]$ being the coefficient of order $i-1$ (lists start at index 1 in MuPAD).

Karatsuba's multiplication. This procedure returns the full product corresponding to the two input list F and G, each containing at least n coefficients. The output is a list of $2n-1$ coefficients.

```
karamul := proc(F,G,n) local fn,gn,F0,G0,F1,G1,i,R,B,nn,x,dom;
begin
   if n=1 then [F[1]*G[1]]
   else
      nn:=(n+1) div 2;
      F0:=[F[i]$i=1..nn]; F1:=[F[i]$i=nn+1..n, 0];
      G0:=[G[i]$i=1..nn]; G1:=[G[i]$i=nn+1..n, 0];
      F:=karamul(F0,G0,nn); G:=karamul(F1,G1,n-nn);
      R:=karamul(zip(F0,F1,_plus),zip(G0,G1,_plus),nn);
      R:=zip(R,zip(F,append(G,0),_plus),_subtract);
      F:=append(F,0) . G;
      [F[i]$i=1..nn, F[nn+i]+R[i]$i=1..n-1, F[i]$i=n+nn..2*n-1]
    end_if;
end_proc:
```

Algorithm `RMP`**.** The following procedure implements Algorithm `RMP` on the input lists A and x of at least $2n-1$ and n elements respectively. The output is a list l of n elements, with $l_i = \sum_{j=1}^{n} A_{n+i-j} x_j$ for $1 \leq i \leq n$.

```
RMP := proc(A,x,n) local a, b, c, k, n2, hi, mi, lo, xhi, xlo, l;
begin
   if n=1 then [A[1]*x[1]]
   elif n mod 2 = 1 then
      if costR(n-1)+2*n-1<costR(n+1) then
         lo:=map([A[k]$k=n..2*n-2], _mult, x[1]);
         a:=_plus(A[2*n-k]*x[k] $ k=1..n);
         append(zip(RMP(A,subsop(x,1=null()),n-1), lo, _plus), a)
      else
         x:=(if nops(x)<=n then append(x,0) else subsop(x,n+1=0) end_if);
         subsop(RMP([0,0] . A, x, n+1), 1=null())
      end_if
   else
      n2:=n/2;
      hi:=[A[k]$k=1..n-1]; mi:=[A[k]$k=n2+1..n+n2-1]; lo:=[A[k]$k=n+1..2*n-1];
      xhi:=[x[k] $ k=1..n2]; xlo:=[x[k] $ k=n2+1..n];
      a:=RMP(zip(hi,mi,_plus), xlo, n2);
```

```
      c:=RMP(zip(mi,lo,_plus), x, n2);
      b:=RMP(mi, zip(xhi,xlo,_subtract), n2);
      zip(a, b, _plus) . zip(c, b, _subtract)
   end_if
end_proc:
```

Algorithm Divide. This procedure returns the quotient of the two input lists A and B, each containing at least n coefficients. The output is a list of n coefficients. B_1 is assumed to be invertible. If the optional fourth argument `opt` equals `"Sqrt"`, then the upper part of the quotient is added to the divisor before computing the second part of the former; this particular form is used from the procedure `Sqrt` below.

```
Divide := proc(A, B, n, opt) local Q,r,i,nhi,nlo;
begin
   if n=1 then [A[1]/B[1]]
   elif n mod 2 = 1 then
      nhi:=(n+1)/2; nlo:=(n-1)/2;
      Q:=Divide(A, B, nhi);
      if opt="Sqrt" then B:=zip(B,[0$nops(B)-n] . Q . [0$nlo],_plus) end_if;
      if costR(nhi)<costR(nlo)+nlo then
         r:=subsop(RMP(B, Q, nhi), 1=null());
      else
         r:=RMP(subsop(B, 1=null(), 2=null()), Q, nlo);
         r:=[r[i]+B[1+i]*Q[nhi] $ i=1..nlo];
      end_if;
      Q . Divide([A[nhi+i]-r[i] $ i=1..nlo], B, nlo);
   else
      Q:=Divide(A, B, n/2);
      if opt="Sqrt" then B:=zip(B,[0$nops(B)-n] . Q . [0$n/2],_plus) end_if;
      r:=RMP(subsop(B,1=null()), Q, n/2);
      Q . Divide([A[n/2+i]-r[i] $ i=1..n/2], B, n/2);
   end_if
end_proc:
```

Algorithm DivRem. The following two procedures `DivRem` and `Div3by2` implements Burnikel and Ziegler's Karatsuba's division algorithm. `DivRem` expects two lists A and B of $2n-1$ and n elements respectively, and returns a quotient list of n elements and a remainder list of $n-1$ elements.

```
DivRem := proc(A, B, n) local n2, i, Q, R;
begin
   if n=1 then [A[1]/B[1]],[]
   elif n mod 2 = 0 then
      n2:=n/2; Q:=Div3by2(A, B, n2);
      R:=Div3by2(Q[2] . [A[i] $ i=n+n2..2*n-1], B, n2);
      Q[1] . R[1], R[2]
   else
```

```
      n2:=(n+1)/2;
      R:=(if nops(B)<=n then append(B, 0) else subsop(B,n+1=0) end_if);
      Q:=Div3by2(A, R, n2);
      R:=Div3by2(Q[2] . [A[i]$i=3*n2..2*n-2], B, n2-1);
      Q := Q[1] . R[1];
      R := R[1], append(R[2], A[2*n-1]);
      Q, [R[2][i] $ i=1..n2-1, R[2][n2-1+i]-B[n]*R[1][i] $ i=1..n2-1];
   end_if
end_proc:

Div3by2 := proc(A, B, n) local Q, i, R;
begin
   Q:=DivRem(A, B, n);
   R:=karamul(Q[1], [B[i]$i=n+1..2*n], n);
   Q[1], [Q[2][i]-R[i]$i=1..n-1, A[n+i]-R[i] $ i=n..2*n-1];
end_proc:
```

Algorithm SqrtRem.

```
SqrtRem := proc(A,n) local S, Q, i, n2, R, Ahi, Alo;
begin
   if n=1 then [sqrt(A[1])],[]
   else
      n2:=(n+1) div 2; Ahi:=[A[i]$i=1..2*n2-1]; Alo:=[A[i]$i=2*n2..2*n-1];
      S:=SqrtRem(Ahi, n2);
      Q:=DivRem(S[2] . Alo, map(S[1],_mult,2), n-n2);
      R:=karamul(Q[1], Q[1], n-n2);
      if 2*n2>n then R:=[0] . R end_if;
      R:=zip(Q[2] . [0$n2], R, _subtract);
      if 2*n2>n then
         R:=[R[i]-2*Q[1][i]*S[1][n2]$i=1..n2-1, R[i]$i=n2..n-1];
      end_if;
      R:=[R[i]$i=1..n-1-n2, R[n-1-n2+i]+Alo[2*n-3*n2+i]$i=1..n2];
      S[1] . Q[1], R
   end_if
end_proc:
```

Algorithm Sqrt.

```
Sqrt := proc(A, n) local n2, S, R, T, i;
begin
   if n=1 then [sqrt(A[1])]
   else
      n2 := (n+2) div 3;
      S := SqrtRem(A, n2);
      R := append(S[2], A[i]$i=2*n2..n, 0$n-1);
      T := map(S[1], _mult, 2) . [0$n-n2];
```

```
      R := Divide(R, T, n-n2, "Sqrt");
      S[1] . R
   end_if
end_proc:
```

References

[1] 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.

[2] Karatsuba, A. A., and Ofman, Y. P. Multiplication of multiplace numbers by automata. *Dokl. Akad. Nauk SSSR 145*, 2 (1962), 293–294.

[3] Karp, A. H., and Markstein, P. High precision division and square root. HP Labs Report 93-93-42, Hewlett Packard, June 1993. Revised October 1994.

[4] Mulders, T. On short multiplications and division. ETH Zurich, 1998. submitted to AAECC.

[5] Quercia, M. Chronométrage d'algorithmes multiprécision. Unpublished document. Available from `http://pauillac.inria.fr/~quercia/papers/mesures.tar.gz`

[6] van der Hoeven, J. Relax, but don't be too lazy. Tech. Rep. 78, Université de Paris-Sud, Mathématiques, Nov. 1999.

[7] Zimmermann, P. Karatsuba square root. Tech. Rep. 3805, INRIA, Nov. 1999.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399