A Rigorous Extension of the Schönhage-Strassen Integer Multiplication Algorithm Using Complex Interval Arithmetic

Thomas Steinke
Department of Mathematics and Statistics
University of Canterbury
Christchurch, New Zealand
tas74@student.canterbury.ac.nz

Raazesh Sainudiin
Department of Mathematics and Statistics
University of Canterbury
Christchurch, New Zealand
r.sainudiin@math.canterbury.ac.nz

Multiplication of n-digit integers by long multiplication requires $O(n^2)$ operations and can be time-consuming. In 1970 A. Schönhage and V. Strassen published an algorithm capable of performing the task with only $O(n\log(n))$ arithmetic operations over $\mathbb{C}$; naturally, finite-precision approximations to $\mathbb{C}$ are used and rounding errors need to be accounted for. Overall, using variable-precision fixed-point numbers, this results in an $O(n(\log(n))^{2+\varepsilon})$-time algorithm. However, to make this algorithm more efficient and practical we need to make use of hardware-based floating-point numbers. How do we deal with rounding errors? and how do we determine the limits of the fixed-precision hardware? Our solution is to use interval arithmetic to guarantee the correctness of results and determine the hardware's limits. We examine the feasibility of this approach and are able to report that 75,000-digit base-256 integers can be handled using double-precision containment sets. This clearly demonstrates that our approach has practical potential; however, at this stage, our implementation does not yet compete with commercial ones, but we are able to demonstrate the feasibility of this technique.

1 Introduction

Multiplication of very large integers is a crucial subroutine of many algorithms such as the RSA cryptosystem [7]. Consequently, much effort has gone into finding fast and reliable multiplication algorithms; [4] discusses several methods. The asymptotically-fastest known algorithm [1] requires $n\log(n)2^{O(\log^*(n))}$ steps, where $\log^*$ is the iterated logarithm — defined as the number of times one has to repeatedly take the logarithm before the number is less than 1. However, being asymptotically-faster does not translate to being faster in practice. We shall concern ourselves with the practicalities of the subject; we will analyse our algorithm's performance on a finite range of numbers.

The algorithm we are studying here is based on the first of two asymptotically fast multiplication algorithms by A. Schönhage and V. Strassen [8]. These algorithms are based on the convolution theorem and the fast Fourier transform. The first algorithm (the one we are studying) performs the discrete Fourier transform over $\mathbb{C}$ using finite-precision approximations. The second algorithm uses the same ideas as the first, but it works over the finite ring $\mathbb{Z}_{2^{2^n}+1}$ rather than the uncountable field $\mathbb{C}$. We wish to point out that "*the* Schönhage-Strassen algorithm" usually refers to the second algorithm. However, in this document we use it to refer to the first $\mathbb{C}$-based algorithm.

From the theoretical viewpoint, the second algorithm is much nicer than the first. The second algorithm does not require the use of finite-precision approximations to $\mathbb{C}$. Also, the second algorithm requires $O(n\log(n)\log(\log(n)))$ steps to multiply two n-bit numbers, making it asymptotically-faster than the first algorithm. However, the second algorithm is much more complicated than the first, and it is outperformed by asymptotically-slower algorithms, such as long multiplication, for small-to-medium input sizes. In practice, both of the Schönhage-Strassen algorithms are rarely used.

Submitted to:
X. Zheng and N. Zhong (Eds.)
Computability and Complexity in Analysis (CCA 2010)

© T. Steinke & R. Sainudiin
This work is licensed under the
Creative Commons Attribution License.

The first Schönhage-Strassen Algorithm is more elegant, if the finite-precision approximations are ignored. More importantly, it is faster in practice. Previous studies [2] have shown that the first algorithm can be faster than even highly-optimised implementations of the second. However, the first algorithm's reliance on finite-precision approximations, despite exact answers being required, leads to it being discounted.

The saving grace of the Schönhage-Strassen algorithm is that at the end of the computation an integral result will be obtained. So the finite-precision approximations are rounded to integers. Thus, as long as rounding errors are sufficiently small for the rounding to be correct, an exact answer will be obtained. Shönhage and Strassen showed that fixed-point numbers with a variable precision of $O(\log(n))$ bits would be sufficient to achieve this.

For the Schönhage-Strassen algorithm to be practical, we need to make use of hardware-based floating-point numbers; software-based variable-precision numbers are simply too slow. However, we need to be able to manage the rounding errors. At the very least, we must be able to detect when the error is too large and more precision is needed. The usual approach to this is to prove some kind of worst-case error bound (for an example, see [6]). Then we can be sure that, for sufficiently small inputs, the algorithm will give correct results. However, worst-case bounds are rarely tight. We propose the use of dynamic error bounds using existing techniques from computer-aided proofs.

Dynamic error detection allows us to move beyond worst-case bounds. For example, using standard single-precision floating-point numbers, our naïve implementation of the Schönhage-Strassen algorithm sometimes gave an incorrect result when we tried multiplying two 120-digit base-256 numbers, but it usually gave correct results. Note that by a 'naïve implementation' we simply mean a modificiation of the Schönhage-Strassen algorithm that uses fixed-precison floating-point arithmetic and does not guarantee correctness. A worst-case bound would not allow us to use the algorithm in this case, despite it usually being correct. Dynamic error detection, however, would allow us to try the algorithm, and, in the rare instances where errors occur, it would inform us that we need to use more precision.

We will use complex interval containment sets for all complex arithmetic operations. This means that at the end of the computation, where we would ordinarily round to the nearest integer, we simply choose the unique integer in the containment set. If the containment set contains multiple integers, then we report an error. This rigorous extension of the Schönhage-Strassen algorithm therefore constitutes a computer-aided proof of the desired product. When an error is detected, we must increase the precision being used or we must use a different algorithm.

For those unfamiliar with the Schönhage-Strassen algorithm or with interval arithmetic, we describe these in section 2. Then, in section 3, we show the empirical results of our study. Section 4, our conclusion, briefly discusses the implications of our results.

2 The Algorithm

For the sake of completeness we explain the Schönhage-Strassen algorithm, as it is presented in [8]. We also explain how we have modified the algorithm using interval arithmetic in subsection 2.5. Those already familiar with the material may skip all or part of this section.

We make the convention that a positive integer x is represented in base b (usually $b = 2^k$ for some $k \in \mathbb{N}$) as a vector $x \in \mathbb{Z}_b^n := \{0, 1, 2, \ldots, b-1\}^n$; the value of x is

$$x = \sum_{i=0}^{n-1} x_i b^i.$$

2.1 Basic Multiplication Algorithm

The above definition immediately leads to a formula for multiplication. Let x and y be positive integers with representations $x \in \mathbb{Z}_b^n$ and $y \in \mathbb{Z}_b^m$. Then

$$xy = \left(\sum_{i=0}^{n-1} x_i b^i\right)\left(\sum_{j=0}^{m-1} y_j b^j\right) = \sum_{i=0}^{n+m-2} \sum_{j=\max\{0,i-m+1\}}^{\min\{n-1,i\}} x_j y_{i-j} b^i = \sum_{i=0}^{n+m-1} z_i b^i.$$

Of course, we cannot simply set $z_i = \sum_{j=\max\{0,i-m+1\}}^{\min\{n-1,i\}} x_j y_{i-j}$; this may violate the constraint that $0 \le z_i \le b-1$ for every i. We must 'carry' the 'overflow'. This leads to the long multiplication algorithm (see [4]).

The Long Multiplication Algorithm

```
1.   Input:  x ∈ Z_b^n and y ∈ Z_b^m
2.   Output:  z ∈ Z_b^{n+m}  # z = xy
3.   Set c = 0.  # c = carry
4.   For i = 0 up to n+m−1 do {
5.        Set s = 0.  # s = sum
6.        For j = max{0, i−m+1} up to min{n−1, i} do {
7.             Set s = s + x_j y_{i−j}.
8.        }.
9.        Set z_i = (s+c)  mod  b.
10.       Set c = ⌊(s+c)/b⌋.
11.  }.
12.  # c = 0 at the end.
```

This algorithm requires $O(mn)$ steps (for a fixed b). Close inspection of the long multiplication algorithm might suggest that $O(mn\log(\min\{m,n\}))$ steps are required as the sum s can become very large. However, adding a bounded number ($x_j y_{i-1} < b^2$) to an unbounded number (s) is, *on average*, a constant-time operation.

2.2 The Discrete Fourier Transform

The basis of the Schönhage-Strassen algorithm is the discrete Fourier transform and the convolution theorem. The discrete Fourier transform is a map from $\mathbb{C}^n$ to $\mathbb{C}^n$. In this section we will define the discrete Fourier transform and we will show how it and its inverse can be calculated with $O(n\log(n))$ complex additions and multiplications. See [4] for further details.

Definition 1 (Discrete Fourier Transform). *Let $x \in \mathbb{C}^n$ and let $\omega := e^{\frac{2\pi i}{n}}$. Then define the discrete Fourier transform $\hat{x} \in \mathbb{C}^n$ of x by*

$$\hat{x}_i := \sum_{j=0}^{n-1} x_j \omega^{ij} \quad (0 \le i \le n-1).$$

There is nothing special about our choice of ω; the consequences of the following lemma are all that we need ω to satisfy. Any other element of $\mathbb{C}$ with the same properties would suffice.

Lemma 2. *Let $n > 1$ and $\omega = e^{\frac{2\pi i}{n}}$. Then*

$$\omega^n = 1 \text{ and } \omega^k \ne 1 \text{ for all } 0 < k < n$$

and, for all $0 < k < n$*,*

$$\sum_{i=0}^{n-1} \omega^{ik} = 0.$$

Note that the case where $n = 1$ is uninteresting, as $\omega = 1$ and the discrete Fourier transform is the identity mapping $\hat{x} = x$.

Proof. Firstly,

$$\omega^n = \left(e^{\frac{2\pi i}{n}}\right)^n = e^{2\pi i} = 1.$$

We know that $e^\theta = 1$ if and only if $\theta = 2\pi i m$ for some $m \in \mathbb{Z}$. Thus, if $\omega^k = 1$, then k must be a multiple of n, which eliminates the possibility that $0 < k < n$.

Fix k with $0 < k < n$ and let $s_k := \sum_{i=0}^{n-1} \omega^{ik}$. Then

$$\omega^k s_k = \sum_{i=0}^{n-1} \omega^{(i+1)k} = \sum_{i=1}^{n} \omega^{ik} = \sum_{i=1}^{n-1} \omega^{ik} + \omega^{kn} = \sum_{i=1}^{n-1} \omega^{ik} + 1 = \sum_{i=1}^{n-1} \omega^{ik} + \omega^{0k} = \sum_{i=0}^{n-1} \omega^{ik} = s_k.$$

So $\omega^k s_k = s_k$. If $s_k \neq 0$, then we can divide by s_k to get $\omega^k = 1$, which is impossible. So $s_k = 0$. □

Now we can prove that the discrete Fourier transform is a bijection.

Proposition 3 (Inverse Discrete Fourier Transform)**.** *Let* $x \in \mathbb{C}^n$ *and let* $\omega = e^{\frac{2\pi i}{n}}$*. Define* $\check{x} \in \mathbb{C}^n$ *by*

$$\check{x}_i := \frac{1}{n} \sum_{j=0}^{n-1} x_j \omega^{-ij} \quad (0 \leq i \leq n-1).$$

Then this defines the inverse of the discrete Fourier transform — that is, if $y = \hat{x}$*, then* $\check{y} = x$*.*

Proof. Fix $x \in \mathbb{C}^n$, let $y = \hat{x}$ and let $z = \check{y}$. We wish to show that $z = x$. If $n = 1$, then this is trivial, as $x = y = z$, so we may assume that $n > 1$. First of all, it follows from Lemma 2 that, if $l \in \mathbb{Z}$ and n does not divide l, then

$$\sum_{i=0}^{n-1} \omega^{il} = 0.$$

If, on the other hand, n divides l, then

$$\sum_{i=0}^{n-1} \omega^{il} = n.$$

Now, fixing i with $0 \leq i \leq n-1$, we have

$$\begin{aligned} z_i &= \frac{1}{n}\sum_{j=0}^{n-1} y_j \omega^{-ij} = \frac{1}{n}\sum_{j=0}^{n-1}\left(\sum_{k=0}^{n-1} x_k \omega^{jk}\right)\omega^{-ij} = \frac{1}{n}\sum_{k=0}^{n-1}\sum_{j=0}^{n-1} x_k \omega^{jk}\omega^{-ij} = \frac{1}{n}\sum_{k=0}^{n-1} x_k \sum_{j=0}^{n-1}\omega^{j(k-i)} \\ &= \frac{1}{n}\sum_{k=0}^{n-1} x_k \left\{\begin{array}{ll} n, & \text{if } n \text{ divides } k-i \\ 0, & \text{otherwise} \end{array}\right\} = \sum_{k=0}^{n-1} x_k \left\{\begin{array}{ll} 1, & \text{if } k-i=0 \\ 0, & \text{otherwise} \end{array}\right\} \\ &= x_i. \end{aligned}$$

□

Now we explain the fast Fourier transform; this is simply a fast algorithm for computing the discrete Fourier transform and its inverse.

Let n be a power of 2 and $x \in \mathbb{C}^n$ be given. Now define $x_{\text{even}}, x_{\text{odd}} \in \mathbb{C}^{n/2}$ by

$$(x_{\text{even}})_i = x_{2i}, \qquad (x_{\text{odd}})_i = x_{2i+1},$$

for all i with $0 \leq i \leq n/2-1$.

Now the critical observation of the Cooley-Tukey fast Fourier transform algorithm is the following. Fix i with $0 \leq i \leq n-1$ and let $\omega = e^{\frac{2\pi i}{n}}$. Then we have

$$\begin{aligned}
\hat{x}_i &= \sum_{j=0}^{n-1} x_j \omega^{ij} \\
&= \sum_{j=0}^{n/2-1} x_{2j}\omega^{2ij} + \sum_{j=0}^{n/2-1} x_{2j+1}\omega^{2ij+i} \\
&= \sum_{j=0}^{n/2-1} (x_{\text{even}})_j \left(\omega^2\right)^{ij} + \omega^i \sum_{j=0}^{n/2-1} (x_{\text{odd}})_j \left(\omega^2\right)^{ij} \\
&= \sum_{j=0}^{n/2-1} (x_{\text{even}})_j \left(\omega^2\right)^{(i \bmod n/2)j} + \omega^i \sum_{j=0}^{n/2-1} (x_{\text{odd}})_j \left(\omega^2\right)^{(i \bmod n/2)j} \\
&= (\hat{x}_{\text{even}})_{i \bmod n/2} + \omega^i (\hat{x}_{\text{odd}})_{i \bmod n/2}.
\end{aligned}$$

Note that $\left(\omega^2\right)^{n/2} = 1$, so taking the modulus is justified. This observation leads to the following divide-and-conquer algorithm.

The Cooley-Tukey Fast Fourier Transform

```
1.   Input:  n = 2^k and x ∈ C^n
2.   Output:  x̂ ∈ C^n
3.   function FFT(k, x) {
4.        If k = 0, then x̂ = x.
5.        Partition x into x_even, x_odd ∈ C^(n/2).
6.        Compute x̂_even = FFT(k−1, x_even) by recursion.
7.        Compute x̂_odd = FFT(k−1, x_odd) by recursion.
8.        Compute ω = e^(2πi/n).
9.        For i = 0 up to n−1 do {
10.            Set x̂_i = (x̂_even)_(i mod n/2) + ω^i (x̂_odd)_(i mod n/2).
11.       }.
12.  }.
```

It is easy to show that this algorithm requires $O(n \log(n))$ complex additions and multiplications. With very little modification we are also able to obtain a fast algorithm for computing the inverse discrete Fourier transform.

Note that, to compute ω, we can use the recurrence

$$\omega_1 = 1, \;\; \omega_2 = -1, \;\; \omega_4 = \mathrm{i}, \;\; \omega_{2n} = \frac{1+\omega_n}{|1+\omega_n|} \;\; (n \geq 3),$$

where $\omega_n = e^{\frac{2\pi i}{n}}$. Other efficient methods of computing ω are also available.

2.3 The Convolution Theorem

We start by defining the convolution. Let $a, b \in \mathbb{C}^n$. We can interpret a and b as the coefficients of two polynomials — that is,

$$f_a(z) = a_0 + a_1 z + \cdots + a_{n-1} z^{n-1}.$$

The convolution of a and b — denoted by $a * b$ — is, for our purposes, the vector of coefficients obtained by multiplying the polynomials f_a and f_b. Thus we have $f_{a*b}(z) = f_a(z) f_b(z)$ for all $z \in \mathbb{C}$. Note that we can add 'padding zeroes' to the end of the coefficient vectors without changing the corresponding polynomial.

The convolution theorem relates convolutions to Fourier transforms. We only use a restricted form.

Theorem 4 (Convolution Theorem)**.** *Let $a, b \in \mathbb{C}^n$ and $c := a * b \in \mathbb{C}^m$, where $m = 2n-1$. Pad a and b by setting*

$$a' = (a_0, a_1, \cdots, a_{n-1}, 0, \cdots, 0), b' = (b_0, b_1, \cdots, b_{n-1}, 0, \cdots, 0) \in \mathbb{C}^m.$$

Then, for every i with $0 \le i \le m-1$,

$$\hat{c}_i = \hat{a}'_i \hat{b}'_i.$$

Proof. By definition, if $\omega = e^{\frac{2\pi i}{m}}$ and $0 \le i \le m-1$, then

$$\hat{c}_i = f_c(\omega^i) = f_a(\omega^i) f_b(\omega^i) = f_{a'}(\omega^i) f_{b'}(\omega^i) = \hat{a}'_i \hat{b}'_i.$$

□

The convolution theorem gives us a fast method of computing convolutions and, thus, of multiplying polynomials. Given $a, b \in \mathbb{C}^n$, we can calculate $c = a * b$ using only $O(n \log(n))$ arithmetic operations as follows.

The Fast Convolution Algorithm

1. `Input:` $a, b \in \mathbb{C}^n$
2. `Output:` $c = a * b \in \mathbb{C}^m$
3. `Set` $k = \lceil \log_2(2n-1) \rceil$ `and` $m = 2^k$.
4. `# Pad` a `and` b `so they are in` $\mathbb{C}^n$.
5. `Set` $a' = (a_0, a_1, \cdots, a_{n-1}, 0, \cdots, 0), b' = (b_0, b_1, \cdots, b_{n-1}, 0, \cdots, 0) \in \mathbb{C}^m$.
6. `Compute` $\hat{a}' = \texttt{FFT}(k, a')$ `and` $\hat{b}' = \texttt{FFT}(k, b')$.
7. `For` $0 \le i \le m-1$`, set` $\hat{c}_i = \hat{a}'_i \hat{b}'_i$.
8. `Compute` $c = \texttt{FFT}^{-1}(k, \hat{c})$.

2.4 The Schönhage-Strassen Algorithm

The Schönhage-Strassen algorithm multiplies two integers by convolving them and then performing carrys. Let two base-b integer representations be x and y. We consider the digits as the coefficients of two polynomials. Then $x = f_x(b)$, $y = f_y(b)$ and

$$xy = f_x(b) f_y(b) = f_{x*y}(b).$$

So, to compute xy, we can first compute $x*y$ in $O(n\log(n))$ steps and then we can evaluate $f_{x*y}(b)$. The evaluation of $f_{x*y}(b)$ to yield an integer representation z is simply the process of performing carrys.

The Schönhage-Strassen Algorithm

```
1.  Input:  x ∈ Z_b^n and y ∈ Z_b^n
2.  Output:  z ∈ Z_b^{2n} # z = xy
3.  Compute x*y using the fast convolution Algorithm.
4.  Set c = 0.  #carry
5.  For i = 0 up to 2n-2 do {
6.        Set z_i = ((x*y)_i + c) mod b.
7.        Set c = ⌊((x*y)_i + c)/b⌋.
8.  }.
9.  Set z_{2n-1} = c.
```

Clearly the Schönhage-Strassen algorithm performs the multiplication using $O(n\log(n))$ complex arithmetic operations.

When finite-precision complex arithmetic is done, rounding errors are introduced. However, this can be countered: We know that $x*y$ must be a vector of integers. As long as the rounding errors introduced are sufficiently small, we can round to the nearest integer and obtain the correct result. Schönhage and Strassen [8] proved that $O(\log(bn))$-bit floating point numbers give sufficient precision.

2.5 Interval Arithmetic

Our rigorous extension of the algorithm uses containment sets. By replacing all complex numbers with complex containment sets, we can modify the Schönhage-Strassen algorithm to find a containment set of $x*y$; if the containment set only contains one integer-valued vector, then we can be certain that this is the correct value. We have used rectangular containment sets of machine-representable floating-point intervals with directed rounding to guarantee the desired integer product. A brief overview of the needed interval analysis [5] is given next.

Let $\underline{x},\overline{x}$ be real numbers with $\underline{x} \leq \overline{x}$. Let $[\underline{x},\overline{x}] = \{x \in \mathbb{R} : \underline{x} \leq x \leq \overline{x}\}$ be a closed and bounded real interval and let the set of all such intervals be $\mathbb{IR} = \{[\underline{x},\overline{x}] : \underline{x} \leq \overline{x}; \underline{x},\overline{x} \in \mathbb{R}\}$. Note that $\mathbb{R} \subset \mathbb{IR}$ since we allow thin or punctual intervals with $\underline{x} = \overline{x}$. If $\star$ is one of the arithmetic operators $+, -, \cdot, /$, we define arithmetic over operands in $\mathbb{IR}$ by $[\underline{a},\overline{a}] \star [\underline{b},\overline{b}] := \{a \star b : a \in [\underline{a},\overline{a}], b \in [\underline{b},\overline{b}]\}$, with the exception that $[\underline{a},\overline{a}]/[\underline{b},\overline{b}]$ is undefined if $0 \in [\underline{b},\overline{b}]$. Due to continuity and monotonicity of the operations and compactness of the operands, arithmetic over $\mathbb{IR}$ is given by real arithmetic operations with the bounds:

$$
\begin{aligned}
[\underline{a},\overline{a}] + [\underline{b},\overline{b}] &= [\underline{a}+\underline{b}, \overline{a}+\overline{b}] \\
[\underline{a},\overline{a}] - [\underline{b},\overline{b}] &= [\underline{a}-\overline{b}, \overline{a}-\underline{b}] \\
[\underline{a},\overline{a}] \cdot [\underline{b},\overline{b}] &= [\min\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}] \\
[\underline{a},\overline{a}] / [\underline{b},\overline{b}] &= [\underline{a},\overline{a}] \cdot [1/\overline{b}, 1/\underline{b}], \text{ if } 0 \notin [\underline{b},\overline{b}] \quad .
\end{aligned}
$$

In addition to the above elementary operations over elements in $\mathbb{IR}$, our algorithm requires us to contain the range of the square root function over elements in $\mathbb{IR} \cap [0,\infty)$. Once again, due to the monotonicity of the square root function over non-negative reals it suffices to work with the real image of the bounds

$\sqrt{[\underline{x},\overline{x}]} = [\sqrt{\underline{x}}, \sqrt{\overline{x}}]$, if $0 \le \underline{x}$. To complete the requirements for our rigorous extension of the Schönhage-Strassen algorithm we need to extend addition, multiplication and division by a non-zero integer to elements in

$$\mathbb{IC} := \left\{ [\underline{z},\overline{z}] := [\underline{z_1},\overline{z_1}] + \mathrm{i}[\underline{z_2},\overline{z_2}] : [\underline{z_1},\overline{z_1}], [\underline{z_2},\overline{z_2}] \in \mathbb{IR} \right\}.$$

Interval arithmetic over $\mathbb{IR}$ naturally extends to $\mathbb{IC}$, the set of rectangular complex intervals. Addition and subtraction over $[\underline{z},\overline{z}], [\underline{w},\overline{w}] \in \mathbb{IC}$ given by

$$[\underline{z},\overline{z}] \pm [\underline{w},\overline{w}] = ([\underline{z_1},\overline{z_1}] \pm [\underline{w_1},\overline{w_1}]) + \mathrm{i}([\underline{z_2},\overline{z_2}] \pm [\underline{w_2},\overline{w_2}])$$

are sharp but not multiplication or division due to rectangular wrapping effects. Complex interval multiplication and division of a complex interval by a non-negative integer can be contained with real interval multiplications given by

$$[\underline{z},\overline{z}] \cdot [\underline{w},\overline{w}] = ([\underline{z_1},\overline{z_1}] \cdot [\underline{w_1},\overline{w_1}] - [\underline{z_2},\overline{z_2}] \cdot [\underline{w_2},\overline{w_2}]) + \mathrm{i}([\underline{z_1},\overline{z_1}] \cdot [\underline{w_2},\overline{w_2}] + [\underline{z_2},\overline{z_2}] \cdot [\underline{w_1},\overline{w_1}]).$$

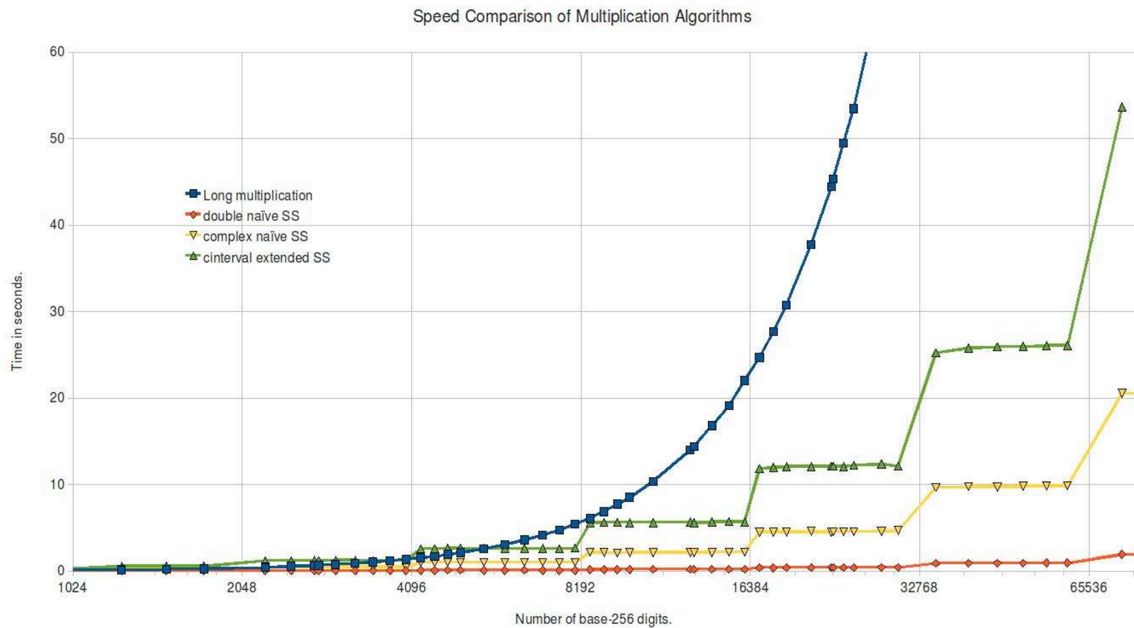
See [3] for details about how C-XSC manipulates rectangular containment sets over $\mathbb{IR}$ and $\mathbb{IC}$.

3 Results

We have implemented the Schönhage-Strassen algorithm, our containment-set version with rectangular complex intervals and long multiplication in C++ using the C-XSC library [3]. Our implementation is available at `http://www.math.canterbury.ac.nz/~r.sainudiin/codes/capa/multiply/` [1]. Results show that, using base 256, our version of the algorithm is usually able to guarantee correct answers for up to 75,000-digit numbers.

The following graph compares the speed of long multiplication (labelled ‘Long multiplication’), the conventional Schönhage-Strassen algorithm with different underlying data types (the implementation using the C-XSC `complex` data type is the line labelled ‘complex naïve SS’ and the one using our own implementation of complex numbers based on the C++ `double` data type is labelled ‘double naïve SS’) and our containment-set version (‘cinterval extended SS’) on uniformly-random n-digit base-256 inputs. All tests were performed on a 2.2 GHz 64-bit AMD Athlon 3500+ Processor running Ubuntu 9.04 using C-XSC version 2.2.4 and gcc version 4.3.1. Times were recorded using the C++ `clock()` function — that is to say, CPU time was recorded. Note that only the ‘Long multiplication’ and ‘cinterval extended SS’ implementations are guaranteed to produce correct results. The ‘double naïve SS’ and ‘complex naïve SS’ implementations may have produced erroneous results, as the implementations do not necessarily provide sufficient precision; these are still shown for comparison. Note also that by ‘naïve’ we mean that these implementations use fixed-precision floating-point arithmetic, whereas the ‘real’ Schönhage-Strassen algorithm uses variable-precision, which is much slower.

[1]Please also download the C-XSC library from `http://www.math.uni-wuppertal.de/~xsc/`.



The above graph shows that the Schönhage-Strassen algorithm is much more efficient than long multiplication for large inputs. However, our modified version of the algorithm is slower than the naïve Schönhage-Strassen algorithm. We believe that `C-XSC` is not well-optimised; for example, their punctual `complex` data type (used in the 'complex naïve SS' implementation) is much slower than our `double`-based complex data type (used in the 'double naïve SS' implementation), even though ostensibly they are the same thing. We see that the `C-XSC cinterval` type (used in the 'cinterval extended SS' implementation) is about three times as slow as the `complex` type. This leaves the possibility that a more optimised implementation of containment sets would be able to compete with commercial algorithms. Investigations such as [2] have shown that the Naïve Schönhage-Strassen algorithm is able to compete with commercial implementations.

Note that the "steps" seen in the graph can be explained by the fact that the algorithm will always round the size up to the nearest power of two. Thus there are steps at the powers of two. The most important feature of our results is the range of input sizes for which our algorithm successfully determines the answer. Using only standard double-precision IEEE floating-point numbers, we are able to use the algorithm to multiply 75,000-digit, or 600,000-bit, integers; this range is more than sufficient for most applications, and at this point the second Schönhage-Strassen algorithm will become competitive.

4 Conclusion

Our investigation has demonstrated that the Schönhage-Strassen algorithm with containment sets is a practical algorithm that could be used reliably for applications such as cryptography. Schöhage and Strassen never showed that their algorithm had any practical value. However, as our implementation was not optimised, this is more of a feasibility study than a finished product.

Note that the advantage of our algorithm over the original Schönhage-Strassen algorithm is that we make use of hardware-based floating-point arithmetic, whereas the original is designed to make use of much slower software-based arithmetic. Both the original algorithm and our adaptation always produce

correct results. However, we use a different approach to guaranteeing correctness. The naïve algorithms we mention are not guaranteed to be correct because they are modifications of the Schönhage-Strassen algorithm which does not take measures to ensure correctness — they simply use fixed-precision floating-point arithemtic and hope for the best; these are only useful for speed comparisons.

It remains to optimise our implementation of the algorithm to compete with commercial libraries. This is dependent on a faster implementation of interval arithmetic. It may also be interesting to use circular containment sets rather than rectangular containment sets. The advantage of circular containment sets is that they easily deal with complex rotations — that is, multiplying by $e^{i\theta}$; this is in fact the only type of complex multiplication (other than division by an integer) that our algorithm performs.

References

[1] Martin Fürer (2007): *Faster Integer Multiplication*. In: *39th ACM STOC*, San Diego, California, USA, pp. 57–66.

[2] Pierrick Gaudry, Alexander Kruppa & Paul Zimmermann (2007): *A gmp-based implementation of schönhage-strassen's large integer multiplication algorithm*. In: *ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, ACM, New York, NY, USA, pp. 167–174.

[3] Hofschuster & Krämer (2004): *C-XSC 2.0: A* `C++` *library for extended scientific computing*. In: R Alt, A Frommer, RB Kearfott & W Luther, editors: *Numerical software with result verification*, *Lecture notes in computer science* 2991, Springer-Verlag, pp. 15–35.

[4] Donald E. Knuth (1998): *The Art of Computer Programming*, 2. Addison-Wesley, 3 edition.

[5] Ramon E. Moore, R. Baker Kearfott & Michael J. Cloud (2009): *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[6] Colin Percival (2003): *Rapid multiplication modulo the sum and difference of highly composite numbers*. *Math. Comput.* 72(241), pp. 387–395.

[7] R.L. Rivest, A. Shamir & L. Adleman (1977): *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. *Communications of the ACM* 21(2), pp. 120–126.

[8] A. Schönhage & V. Strassen (1971): *Schnelle Multiplikation großer Zahlen (Fast Multiplication of Large Numbers)*. *Computing: Archiv für elektronisches Rechnen (Archives for electronic computing)* 7, pp. 281–292. (German).