

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

Thomas Steinke

Raazesh Sainudiin

Department of Mathematics and Statistics  
University of Canterbury  
Christchurch, New Zealand

tas74@student.canterbury.ac.nz

r.sainudiin@math.canterbury.ac.nz

- Somewhat applied...
- ...but some nice ideas.

Big picture:

- We have a theoretically-great algorithm.
- The algorithm requires real computation.
- In practice the algorithm isn't so fantastic.
- We use interval arithmetic to “fix” the real computation problems...
- ...to get a “better” algorithm.

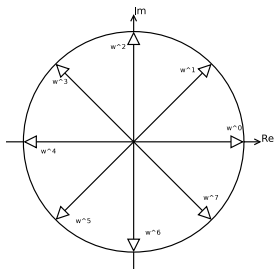
- Motivation & Background for fast multiplication
- Discrete Fourier Transform
- Fast Fourier Transform
- Polynomial Multiplication
- The Schönhage-Strassen Algorithm
- Containment Sets
- Conclusion
- Further Work

# Motivation & Background

- Multiplying large numbers can be time-consuming if done often enough.
- Computers work with 1024-bit numbers regularly for cryptography.
- Long multiplication takes  $O(n^2)$ -time to multiply two  $n$ -bit numbers.
- In 1952 A. Kolmogorov conjectured that there is no faster algorithm than  $O(n^2)$ .
- In 1960 A. Karatsuba found an  $O(n^{\log_2(3)}) = O(n^{1.58})$ -time algorithm.
- **In 1970 A. Schönhage and V. Strassen find an  $O(n \log(n)^3)$ -time algorithm.** They also find an  $O(n \log(n) \log(\log(n)))$ -time algorithm.
- In 2007 M. Fürer found an  $n \log(n) 2^{O(\log^*(n))}$ -time algorithm.

# The Discrete Fourier Transform

- Maps  $\mathbb{C}^n$  to  $\mathbb{C}^n$ .
- Primitive roots of unity.  $\omega_n = e^{\frac{2\pi i}{n}}$ .



- Given a vector  $\mathbf{a} = (a_0, a_1, a_2, \dots, a_{n-1}) \in \mathbb{C}^n$ .
- Interpret  $\mathbf{a}$  as a polynomial  
$$f_{\mathbf{a}}(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}.$$
- The discrete Fourier transform (DFT) of  $\mathbf{a}$  is  
$$\hat{\mathbf{a}} = (f_{\mathbf{a}}(\omega_n^0), f_{\mathbf{a}}(\omega_n^1), f_{\mathbf{a}}(\omega_n^2), \dots, f_{\mathbf{a}}(\omega_n^{n-1})).$$
- The DFT can be represented by a matrix.

# The Fast Fourier Transform 1

Suppose that  $n$  is even and  $\mathbf{a} \in \mathbb{C}^n$ .

$$\begin{aligned}\hat{a}_i &= f_{\mathbf{a}}(\omega_n^i) \\ &= a_0\omega_n^{0i} + a_1\omega_n^{1i} + \cdots + a_{n-1}\omega_n^{(n-1)i} \\ &= \left( a_0(\omega_n^2)^{0i} + a_2(\omega_n^2)^{1i} + \cdots + a_{n-2}(\omega_n^2)^{(n/2-1)i} \right) \\ &\quad + \omega_n^i \left( a_1(\omega_n^2)^{0i} + a_3(\omega_n^2)^{1i} + \cdots + a_{n-1}(\omega_n^2)^{(n/2-1)i} \right)\end{aligned}$$

Let

$$\begin{aligned}\mathbf{a}_{\text{even}} &= (a_0, a_2, \dots, a_{n-2}) \in \mathbb{C}^{n/2}, \\ \mathbf{a}_{\text{odd}} &= (a_1, a_3, \dots, a_{n-1}) \in \mathbb{C}^{n/2}.\end{aligned}$$

Then

$$\hat{a}_i = (\hat{\mathbf{a}}_{\text{even}})_i + \omega_n^i (\hat{\mathbf{a}}_{\text{odd}})_i$$

Using

$$\hat{a}_i = (\hat{\mathbf{a}}_{\text{even}})_i + \omega_n^i (\hat{\mathbf{a}}_{\text{odd}})_i$$

- 'Divide and Conquer' algorithm.  $O(n \log(n))$  complex additions and multiplications needed. ( $n$  must be a power of 2.)
- The inverse discrete Fourier transform is given by  $a_i = \frac{1}{n} f_{\hat{\mathbf{a}}}(\omega_n^{-i})$ .
- The inverse discrete Fourier transform can also be computed using  $O(n \log(n))$  operations.

# Polynomial Multiplication 1

Take two polynomials  $f_{\mathbf{a}}(x)$  and  $f_{\mathbf{b}}(x)$ .

We want to multiply them to get  $f_{\mathbf{c}} = f_{\mathbf{a}} \cdot f_{\mathbf{b}}$ . Then

$$c_i = \sum_j a_j b_{i-j}.$$

But

$$\begin{aligned}\hat{c}_i &= f_{\mathbf{c}}(\omega_n^i) \\ &= f_{\mathbf{a}}(\omega_n^i) f_{\mathbf{b}}(\omega_n^i), \\ &= \hat{a}_i \hat{b}_i.\end{aligned}$$

So we can calculate  $\hat{\mathbf{c}}$  without calculating  $\mathbf{c}$ .



# Polynomial Multiplication 2

This leads to a fast polynomial multiplication algorithm:

- 1 Calculate  $\hat{\mathbf{a}}$  and  $\hat{\mathbf{b}}$  in  $O(n \log(n))$ -time.
- 2 Calculate  $\hat{\mathbf{c}}$  in  $O(n)$ -time.
- 3 Calculate  $\mathbf{c}$  in  $O(n \log(n))$ -time.

Overall, we can multiply in  $O(n \log(n))$ -time – much better than  $O(n^2)$ .

# The Schönhage-Strassen Algorithm 1

Multiplying integers is a special case of multiplying polynomials. Integer multiplication is polynomial multiplication followed by carries.

## The Schönhage-Strassen Algorithm

- 1 Start with two integers expressed as vectors of digits  $\mathbf{a}$  and  $\mathbf{b}$ .
- 2 Choose a suitable  $n$ .
- 3 Compute  $\hat{\mathbf{a}}$  and  $\hat{\mathbf{b}}$  using the fast Fourier transform.
- 4 Compute  $\hat{\mathbf{c}}$  using  $\hat{c}_i = \hat{a}_i \hat{b}_i$ .
- 5 Compute  $\mathbf{c}$  using the inverse fast Fourier transform.
- 6 Perform carries on  $\mathbf{c}$ .
- 7 Now  $\mathbf{c} = \mathbf{a} * \mathbf{b}$ .

# The Schönhage-Strassen Algorithm 2

Problem: Computers cannot do *exact* complex arithmetic.

- Computers can do *approximate* complex arithmetic.
- We know that  $\mathbf{c}$  must be a vector of integers.
- So we round to the nearest integer. e.g.  $1.1 + 0.1i$  is rounded to 1.
- As long as the error is small enough, this will give the correct answer.

## Question

*How can we be sure that the error is small enough for rounding to give the correct answer?*

## Question

*How can we be sure that the error is small enough for rounding to give the correct answer?*

- It can be proved that  $O(\log(n))$ -bit floating point numbers will suffice.
- In practice we have fixed 52-bit (`double`) floating point numbers.
- We can avoid the problem by doing the Fourier transform over a finite ring ( $\mathbb{Z}_{2^{2^n}}$ ). But this is tricky.
- **We can use containment sets to guarantee correct answers.**

# Containment Sets 1

- If  $x \in S$ , then we say  $S$  is a **containment set** (for  $x$ ).
- During computation we may not be able to compute  $f(x)$  exactly but usually we can find a small containment set for  $f(x)$ .
- e.g.  $1/3 \in [0.333, 0.334]$ ;  $\sqrt{2} \in [1.414, 1.415]$ ;  
 $x, y \in [0, 1] \implies x + y \in [0, 2]$  i.e.  $[0, 1] + [0, 1] \subset [0, 2]$
- We can go through the entire polynomial multiplication algorithm this way and compute a containment set for the answer.
- We end up with an interval/rectangle that definitely contains the answer.
- The diameter should be fairly small. It should only contain one integer, which allows us to guarantee the correct outcome.

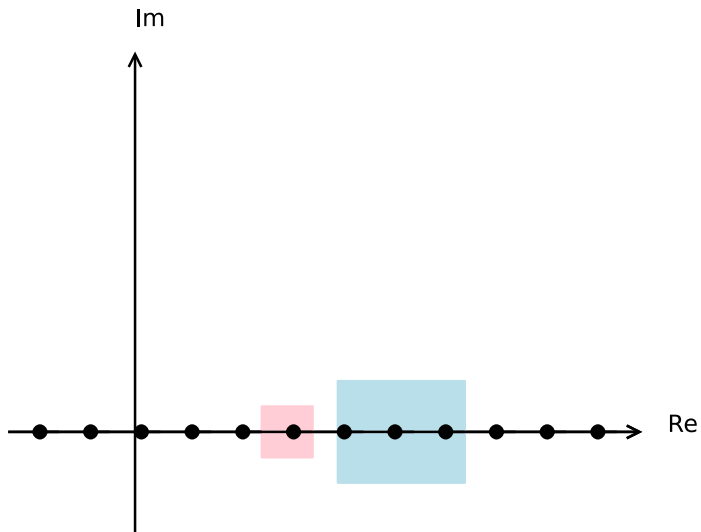
$$[\underline{a}, \bar{a}] + [\underline{b}, \bar{b}] = [\underline{a} + \underline{b}, \bar{a} + \bar{b}]$$

$$[\underline{a}, \bar{a}] - [\underline{b}, \bar{b}] = [\underline{a} - \bar{b}, \bar{a} + \underline{b}]$$

$$[\underline{a}, \bar{a}] \cdot [\underline{b}, \bar{b}] = [\min\{\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}\}]$$

$$[\underline{a}, \bar{a}] / [\underline{b}, \bar{b}] = [\underline{a}, \bar{a}] \cdot [1/\bar{b}, 1/\underline{b}], \text{ if } 0 \notin [\underline{b}, \bar{b}] .$$

# Containment Sets 3

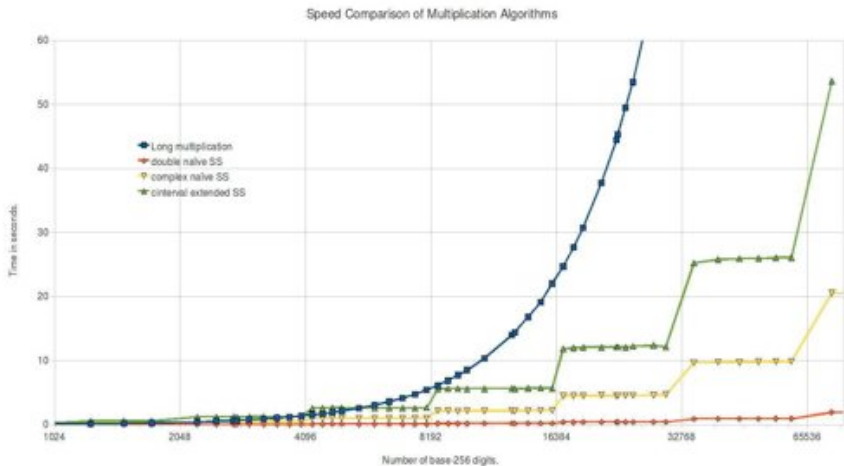


# Conclusion 1

- End result: An algorithm that guarantees its output, unlike a naïve implementation.
- If we cannot guarantee a correct result, we have to increase the precision and try again. (Or use a different algorithm.)
- The speed of this algorithm is a constant multiple of that of the naïve implementation.

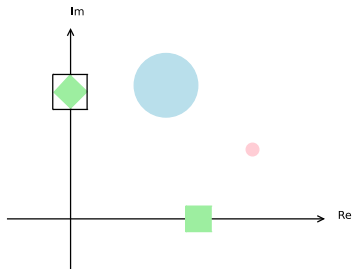


# Conclusion 2



# Further Work

- Results are promising.
- Optimise the implementation to see how it compares to other fast multiplication algorithms.
- More accurate calculation of  $\omega_n = e^{\frac{2\pi i}{n}}$ .
- Circular containment sets – better for rotation/multiplication by  $\omega_n$ .



# The End