DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Algorithms for Large Matrix Multiplications

Assessment of Strassen's Algorithm

BJÖRN JOHANSSON

EMIL ÖSTERBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

EXAMENSARBETE INOM TEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2018*

Algoritmer för Stora Matrismultiplikationer

Bedömning av Strassens Algoritm

BJÖRN JOHANSSON

EMIL ÖSTERBERG

KTH
SKOLAN FÖR TEKNIKVETENSKAP

Abstract

Strassen's algorithm was one of the breakthroughs in matrix analysis in 1968. In this report the thesis of Volker Strassen's algorithm for matrix multiplications along with theories about precisions will be shown. The benefits of using this algorithm compared to naive matrix multiplication and its implications, how its performance compare to the naive algorithm, will be displayed. Strassen's algorithm will also be assessed on how the output differ when the matrix sizes grow larger, as well as how the theoretical complexity of the algorithm differs from the achieved complexity.

The studies found that Strassen's algorithm outperformed the naive matrix multiplication at matrix sizes 1024×1024 and above. The achieved complexity was a little higher compared to Volker Strassen's theoretical. The optimal precision for this case were the double precision, Float64.

How the algorithm is implemented in code matters for its performance. A number of techniques need to be considered in order to improve Strassen's algorithm, optimizing its termination criterion, the manner by which it is padded in order to make it more usable for recursive application and the way it is implemented e.g. parallel computing. Even tough it could be proved that Strassen's algorithm outperformed the Naive after reaching a certain matrix size, it is still not the most efficient one; e.g. as shown with Strassen-Winograd. One need to be careful of how the sub-matrices are being allocated, to not use unnecessary memory. For further reading one can study cache-oblivious and cache-aware algorithms.

Keywords: Strassen, Matrix multiplication, Precision, Complexity

Sammanfattning

1968 var Strassens algoritm en av de stora genombrotten inom matrisanalyser. I denna rapport kommer teorin av Volker Strassens algoritm för matrismultiplikationer tillsammans med teorier om precisioner att presenteras. Även fördelar med att använda denna algoritm jämfört med naiva matrismultiplikation och dess implikationer, samt hur den presterar jämfört med den naiva algoritmen kommer att presenteras. Strassens algoritm kommer också att bli bedömd på hur dess resultat skiljer sig för olika precisioner när matriserna blir större, samt hur dess teoretiska komplexitet skiljer sig gentemot den erhållna komplexiteten.

Studier hittade att Strassens algoritm överträffade den naiva algoritmen för matriser av storlek 1024×1024 och större. Den erhållna komplexiteten var lite större än Volker Strassens teoretiska. Den optimala precisionen i detta fall var dubbelprecisionen, Float64.

Sättet algoritmen implementeras på i koden påverkar dess prestanda. Ett flertal olika faktorer behövs ha i åtanke för att förbättra Strassens algoritm: optimera dess avbrottsvillkor, sättet som matriserna paddas för att de ska vara mer användbara för rekursiv tillämpning och hur de implementeras t.ex. parallella beräkningar. Även om det kunde bevisas att Strassen algoritm överträffade den naiva efter en viss matrisstorlek så är den inte den mest effektiva; t.ex visades detta med Strassen-Winograd. Man behöver vara uppmärksam på hur undermatriserna allokeras, för att inte ta upp onödigt minne. För fördjupning kan man läsa på om cache-oblivious och cache-aware algoritmer.

Sökord: Strassen, Matrismultiplikation, Precision, Komplexitet

Acknowledgments

Our thesis have been under the supervision of Assoc. Prof. Elias Jarlebring and we are thankful for his guidance and patience when explaining new terminology to our knowledge. We would also like to thank the KTH's Dept. of Mathematics for giving us access to their server.

Björn Johansson
bjorn8@kth.se

Emil Österberg
emilost@kth.se

Stockholm, Sweden
May 21, 2018

Contents

1 Introduction

To better understand where we are going, it is necessary to know where we are coming from; where the improvements have been made. We have to understand the foundation of the basic matrix multiplication, how it is done, what operations are needed, how it changed in 1968 and how it can be implemented today.

Given two $m \times m$ matrices $\mathbf{A}$, $\mathbf{B}$ and performing the multiplication $\mathbf{AB} = \mathbf{C}$, the operations needed are m^3 multiplications and $m^2(m-1)$ additions, total of $2m^3 - m^2$ operations. This is the number of operations required to calculate a matrix multiplications using the naive algorithm. When trying to optimize the standard matrix multiplication, one is trying to reduce the number of total operations and thereby the complexity, $\mathcal{O}(m^3)$, of the algorithm. When Strassen introduced his algorithm for matrix multiplication, he had managed to reduce the operations to $\mathcal{O}(m^{\log_2(7)})$. Though at the time of his publication the matrix size at which his algorithm was more efficient than the naive algorithm rendered it quite impractical. How practical it is today will be discussed in detail later on and how it stands against naive matrix multiplication as m becomes finitely large.

The efficiency of Strassen's algorithm in comparison to other algorithms, will be studied and evaluated in this report, along with how stable it is depending on floating point precision. This report will examine how useful Strassen's algorithm is regarding how the complexity rises with growing matrix size, together with the extra memory allocations, and the rounding error due to the large level of arithmetic operations.

Henceforth we will in this report refer to the standard "row-column"-algorithm as Naive, and Strassen's algorithm as Strassen.

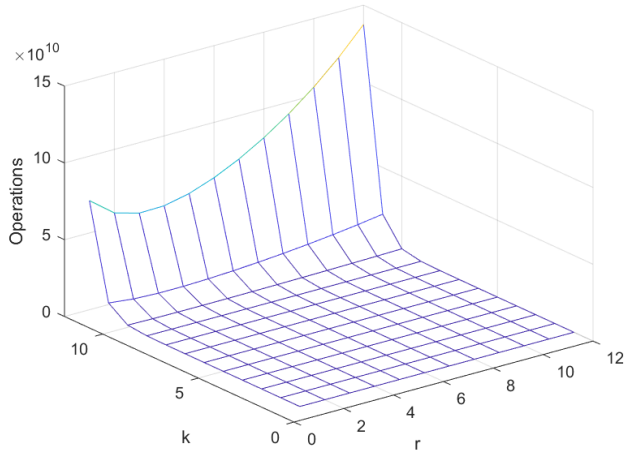


Figure 1: The number of operations required by Strassen's algorithm as a function of matrix size (2^k) and termination criterion (2^r).

2 Strassen's Algorithm

The algorithm separates a square, $m \times m$, matrix into square sub-matrices and trades one multiplication for additions and subtractions which makes it more efficient than the Naive, as multiplication is a more expensive operation. The algorithm, according to Strassen, executes a matrix multiplication with $\leq 4.7m^{\log_2 7}$ operations in comparison with the Naive which gives a complexity of order m^3.

2.1 Method

The algorithm separates the matrices $\mathbf{A}$ and $\mathbf{B}$ of size 2^k into blocks of 2^{k-1} and computes

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad \mathbf{AB} = \mathbf{C} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

and computes

$$\begin{array}{ll} M1 := (A_{11} + A_{22})(B_{11} + B_{22}) & M5 := (A_{11} + A_{12})B_{22} \\ M2 := (A_{21} + A_{22})B_{11} & M6 := (A_{21} - A_{11})(B_{11} + B_{12}) \\ M3 := A_{11}(B_{12} - B_{22}) & M7 := (A_{12} - A_{22})(B_{21} + B_{22}), \\ M4 := A_{22}(B_{21} - B_{11}) & \end{array} \tag{1}$$

which then summarizes into the submatrices

$$\begin{array}{ll} C_{11} &= M1 + M4 - M5 + M7 \\ C_{12} &= M3 + M5 \\ C_{21} &= M2 + M4 \\ C_{22} &= M1 + M3 - M2 + M6. \end{array} \tag{2}$$

These then generate the matrix $\mathbf{C}$

$$\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

More specifically the algorithm will compute the multiplication of square matrices of size $m = 2^k$ by subdividing the matrix into four sub-matrices. At each of the seven multiplications, M_i for $i = 1, 2, .., 7$, when the sub-matrices are to be multiplied Strassen will be re-applied, in a recursive manner. Strassen will keep dividing the matrices into sub-matrices until they reach a termination criterion where the sub-matrices are of a size 2^r, a size where naive matrix multiplication is preferred. Strassen has then been used recursively $k - r$ times [1].

Following is the algorithm, including an implementation of a matrix padding.

Algorithm 1 Strassen's Algorithm

Function strassen(A, B)
$(row_A$,$column_A)$ = size(A) ;
$(row_B$,$column_B)$ = size(B) ;

function PAD(N,Z)
  (*row*,*column*) = size(Z) ;
  if row == N && column == N **then**
    return Z
  end if
  P = zeros(N,N) ;
  P(1:row,1:column) = Z ;
end function

n=2^($ceil(\log_2(\max(row_{A,B}, column_{A,B})))$) ;
A = PAD(n,A) ;
B = PAD(n,B) ;

A_{11} = $A[1 : \frac{row}{2}, 1 : \frac{column}{2}]$; B_{11} = $B[1 : \frac{row}{2}, 1 : \frac{column}{2}]$;
A_{12} = $A[1 : \frac{row}{2}, \frac{column}{2} + 1 : end]$; B_{12} = $B[1 : \frac{row}{2}, \frac{column}{2} + 1 : end]$;
A_{21} = $A[\frac{row}{2} + 1 : end, 1 : \frac{column}{2}]$; B_{21} = $B[\frac{row}{2} + 1 : end, 1 : \frac{column}{2}]$;
A_{22} = $A[\frac{row}{2} + 1 : end, \frac{column}{2} + 1 : end]$; B_{22} = $B[\frac{row}{2} + 1 : end, \frac{column}{2} + 1 : end]$;

if $row_{A,B} \leq 2^r$ && $column_{A,B} \leq 2^r$ **then**
  I= naive($A_{11} + A_{22}$,$B_{11} + B_{22}$) ;
  II = naive($A_{21} + A_{22}$,B_{22}) ;
  III = naive(A_{11},$B_{12} - B_{22}$) ;
  IV = naive(A_{22},$B_{21} - B_{11}$) ;
  V = naive($A_{11} + A_{12}$,B_{22}) ;
  VI = naive($A_{21} - A_{11}$,$B_{11} + B_{12}$) ;
  VII = naive($A_{12} - A_{22}$,$B_{21} + B_{22}$) ;

C_{11} = I + IV - V + VII ;
C_{12} = III + V ;
C_{21} = II + IV ;
C_{22} = I - II + III + VI;

  return C = $\left[C_{11}, \quad C_{12}; C_{21}, \quad C_{22} \right]$
else
  I = strassen($A_{11} + A_{22}$,$B_{11} + B_{22}$) ;
  II = strassen($A_{21} + A_{22}$,B_{22}) ;
  III = strassen(A_{11},$B_{12} - B_{22}$) ;
  IV = strassen(A_{22},$B_{21} - B_{11}$) ;
  V = strassen($A_{11} + A_{12}$,B_{22}) ;
  VI = strassen($A_{21} - A_{11}$,$B_{11} + B_{12}$) ;
  VII = strassen($A_{12} - A_{22}$,$B_{21} + B_{22}$) ;

end if ▷ Strassen's algorithm with power of two-padding

2.2 Implementation

In practice Strassen can be implemented for all shapes of matrices and is not limited to square ones. There are several ways of implementation of the algorithm that are useful, but are requiring more computation power.

To broaden the variety of matrices on which the algorithm is applicable the matrices can be padded[1] [2]. There are different implementations of padding to the algorithm: e.g. power of two-padding, dynamic padding, and dynamic peeling. Some of these require different separations for the sub-matrices dimensions to match.

The simplest implementation of padding is to create the smallest square, $2^n \times 2^n$, null matrix which A and B will fit inside, as shown in the pseudo code in the previous section. This method will also make the algorithm applicable to rectangular matrices. In our terminology we refer to this as Power of two-padding.

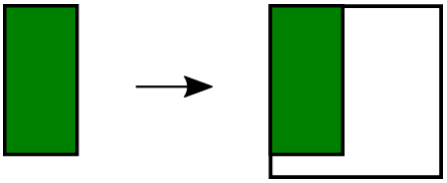


Figure 2: The original matrix being concatenated with the power of two-null matrix. The new matrix being significantly larger, and more memory demanding than the original.

There are obvious disadvantages with this. For larger matrix sizes the null matrix might be very large, allocating a lot of memory thus making it a slower process (figure 2). The memory inefficiency can be solved by a more interactive procedure, called dynamic padding.

Figure 3: The original matrix with a row and/or column added in order to make the algorithm applicable for Strassen.

Dynamic padding here refers to the procedure recognizing if the number of rows or columns is odd and and padding said matrix with a zero row, column, or both (figure

[1]Padding refers to enlarging the matrices with rows and columns with zeros such that the algorithm can be applied, because even though the matrix dimensions agree, the algorithm might not be able to compute the multiplication.

3). Implementation of dynamic padding is more complex than power of two-padding due to possible dimension mismatch when multiplying the sub matrices. Possible solutions can be resolving to the Naive when this occurs, or combining the dynamic- with the power of two-padding for the sub-matrices.

Figure 4: The original matrix with a row and column peeled of, leaving the A_{11} to be applicable for Strassen.

An opposite approach to the padding problem is dynamic peeling (figure 4). Here the matrix is sub-divided, and the superfluous row and column peeled of. The product $\mathbf{C}$ of the multiplication of a $m \times k$ matrix $\mathbf{A}$, and $k \times n$ matrix $\mathbf{B}$ is then calculated as

$$C = \begin{bmatrix} C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} & C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22} \\ C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21} & C_{22} = A_{21} \times B_{11} + A_{22} \times B_{22} \end{bmatrix}, \tag{3}$$

where A_{11} is a $(m-1) \times (k-1)$ matrix, A_{12} is a $(m-1) \times 1$ matrix, A_{21} is a $1 \times (k-1)$ matrix, and A_{22} is a 1×1 matrix, and B_{11} is a $(k-1) \times (n-1)$ matrix, B_{12} is a $(k-1) \times 1$ matrix, B_{21} is a $1 \times (n-1)$ matrix, and B_{22} is a 1×1 matrix. The $A_{11} \times B_{11}$ products are calculated using Strassen, and the other products using naive multiplication [3].

3 Complexity

Whether or not the algorithm is useful, and to understand its behavior, largely depends on the complexity which the algorithm grows with. The complexity, commonly referred to as "big-O" and denoted $\mathcal{O}$, indicates that the time for solving systems of functions or algorithms grows at a certain rate. This notation implies that, for large m, the emphasis is on the terms of m with the highest exponent while the lower exponent terms of m become negligible. Naive matrix multiplication of two $m \times m$ matrices for example requires a total of $m^3 + (m-1)m^2$ arithmetic operations, and as m grows large the m^2-term will account for a very small part of the calculation time. The complexity for the Naive multiplication is thereby $\mathcal{O}(m^3)$, the worst case scenario which the computational time will not grow larger than, as m grows large. More concisely the complexity is a measurement of how the time required to run the elementary operations calculated by the algorithm varies as the input size varies [4].

To theoretically determine the complexity of an algorithm one has to count the operations of said algorithm. If $\mathbf{A}$ is a $m \times n$ matrix and $\mathbf{B}$ is a $n \times p$ matrix Strassen requires $\frac{7}{8}mnp$ multiplications and $\frac{7}{8}m(n-2)p + \frac{5}{4}mn + \frac{5}{4}np + \frac{8}{4}mp$ additions. If we assume that the matrices are square, of the same size and contain 2^k elements, and apply the algorithm recursively until the termination criterion is reached and the block matrices are of size $m_0 = 2^r$ it can be shown that the number of multiplications and additions needed is $7^{k-r}8^r$ multiplications and $4^r(2^r+5)7^{k-r} - 6 \cdot 4^k$ additions and subtractions.

3.1 Justification

The derivation of the operations count for Strassen, and complexity, is as follows. The matrices **A, B**, both of size $m \times m$, are subdivided into four matrices each

$$A_{11} = \begin{bmatrix} a_{1,1} & \dots & a_{1,\frac{m}{2}} \\ \vdots & \ddots & \vdots \\ a_{\frac{m}{2},1} & \dots & a_{\frac{m}{2},\frac{m}{2}} \end{bmatrix} \qquad A_{12} = \begin{bmatrix} a_{1,\frac{m}{2}+1} & \dots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{\frac{m}{2},\frac{m}{2}+1} & \dots & a_{\frac{m}{2},m} \end{bmatrix}$$
$$A_{21} = \begin{bmatrix} a_{\frac{m}{2}+1,1} & \dots & a_{\frac{m}{2}+1,\frac{m}{2}} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,\frac{m}{2}} \end{bmatrix} \quad A_{22} = \begin{bmatrix} a_{\frac{m}{2}+1,\frac{m}{2}+1} & \dots & a_{\frac{m}{2}+1,m} \\ \vdots & \ddots & \vdots \\ a_{m,\frac{m}{2}+1} & \cdots & a_{m,m} \end{bmatrix}, \tag{4}$$

and applied to the algorithm. The algorithm multiplication scheme gives us the equation

$$M1 := (A_{11} + A_{22})(B_{11} + B_{22}) =$$
$$\left(\begin{bmatrix} a_{1,1} & \dots & a_{1,\frac{m}{2}} \\ \vdots & \ddots & \vdots \\ a_{\frac{m}{2},1} & \dots & a_{\frac{m}{2},\frac{m}{2}} \end{bmatrix} + \begin{bmatrix} a_{\frac{m}{2}+1,\frac{m}{2}+1} & \dots & a_{\frac{m}{2}+1,m} \\ \vdots & \ddots & \vdots \\ a_{m,\frac{m}{2}+1} & \cdots & a_{m,m} \end{bmatrix} \right) \tag{5}$$
$$\times \left(\begin{bmatrix} b_{1,1} & \dots & b_{1,\frac{m}{2}} \\ \vdots & \ddots & \vdots \\ b_{\frac{m}{2},1} & \dots & b_{\frac{m}{2},\frac{m}{2}} \end{bmatrix} + \begin{bmatrix} b_{\frac{m}{2}+1,\frac{m}{2}+1} & \dots & b_{\frac{m}{2}+1,m} \\ \vdots & \ddots & \vdots \\ b_{m,\frac{m}{2}+1} & \cdots & b_{m,m} \end{bmatrix} \right),$$

which accounts for $2 \times (\frac{m}{2})^2$ additions and subtractions per matrix addition. It also accounts for $(\frac{m}{2})^3$ multiplications, and a additional $(\frac{m}{2})^2(\frac{m}{2}-1)$ additions per multiplication. By studying this equation we determine that the one level Strassen (without recursion) comprises of seven multiplications along with ten addition and subtractions (equation 1) . When composing these matrices (equation 2) an additional eight additions and subtractions are made. By summarizing all of these arithmetic operations we arrive at

$$\begin{array}{rl} 7 \cdot \frac{m^3}{2^3} & \textit{multiplications} \\ \frac{18}{2^2}m^2 + 7 \cdot \frac{m^2}{2^2}(\frac{m}{2} - 1) & \textit{additions} \end{array} \tag{6}$$

operations. However the desired complexity is achieved when the algorithm is ap-

plied recursively, which means that when each of the seven matrix multiplications are to take place, instead of calculating the matrix multiplication the algorithm is re-applied, sub-dividing the matrices again, and instead of seven multiplications we now have to calculate 7^2 multiplications at the second level and 7^3 at the third, and at the i:th, 7^i. We get the following matrices at the second level

$$\begin{aligned} M1 \coloneqq & \left(\begin{bmatrix} a_{1,1} & \dots & a_{1,\frac{m}{4}} \\ \vdots & \ddots & \vdots \\ a_{\frac{m}{4},1} & \dots & a_{\frac{m}{4},\frac{m}{4}} \end{bmatrix} + \begin{bmatrix} a_{\frac{m}{4}+1,\frac{m}{4}+1} & \dots & a_{\frac{m}{4}+1,\frac{m}{2}} \\ \vdots & \ddots & \vdots \\ a_{\frac{m}{2},\frac{m}{4}+1} & \cdots & a_{\frac{m}{2},\frac{m}{2}} \end{bmatrix} \right) \\ & \times \left(\begin{bmatrix} b_{1,1} & \dots & b_{1,\frac{m}{4}} \\ \vdots & \ddots & \vdots \\ b_{\frac{m}{4},1} & \dots & b_{\frac{m}{4},\frac{m}{4}} \end{bmatrix} + \begin{bmatrix} b_{\frac{m}{4}+1,\frac{m}{4}+1} & \dots & b_{\frac{m}{4}+1,\frac{m}{2}} \\ \vdots & \ddots & \vdots \\ b_{\frac{m}{2},\frac{m}{4}+1} & \cdots & b_{\frac{m}{2},\frac{m}{2}} \end{bmatrix} \right). \end{aligned} \tag{7}$$

This time the matrices are of size $\frac{m}{4} \times \frac{m}{4}$ and this equation accounts for $2 \cdot (\frac{m}{2^2})^2$ additions and subtractions per matrix addition. It also accounts for $(\frac{m}{2^2})^3$ multiplications, and an additional $\frac{m^2}{(2^2)^2}(\frac{m}{2^2} - 1)$ additions and subtractions. Summarizing the number of operations as done for the second level Strassen we arrive at the following expression

$$\begin{array}{ll} 7^2 \cdot \frac{m^3}{(2^2)^3} & \textit{multiplications} \\ 7 \cdot (\frac{18}{(2^2)^2} m^2) + \frac{18}{2^2} m^2 + 7^2 \cdot \frac{m^2}{(2^2)^2}(\frac{m}{2^2} - 1) & \textit{additions.} \end{array} \tag{8}$$

If the original matrices are of the size that the algorithm can get re-applied a third time we get the following number of operations

$$\begin{array}{ll} 7^3 \cdot \frac{m^3}{(2^3)^3} & \textit{multiplications} \\ 7^2 \cdot (\frac{18}{(2^3)^2} m^2) + 7 \cdot (\frac{18}{(2^2)^2} m^2) + \frac{18}{2^2} m^2 + 7^3 \cdot \frac{m^2}{(2^3)^2}(\frac{m}{2^3} - 1) & \textit{additions.} \end{array} \tag{9}$$

If the original matrices are of the size 2^k and we want to run the algorithm, and re-applying it, until the sub-matrices are of the size 2^r we can recognize a pattern. The number of operations for the generalized case we get

$$\begin{array}{c} 7^{(k-r)} \cdot \frac{m^3}{(2^{(k-r)})^3} \\ \frac{7^{(k-r)}}{(2^{(k-r)})^2}(\frac{m}{2^{(k-r)}} - 1)m^2 + \sum_{i=0}^{k-r-1} 7^i (\frac{18}{2^{(i+1)}}) m^2. \end{array} \tag{10}$$

The matrix size, m, is 2^k. Also $\sum_{i=0}^{k-r-1} 7^i (\frac{18}{2^{(i+1)}})(2^k)^2 = \frac{18 \cdot 4^k}{3}\left((\frac{7}{4})^{(k-r)} - 1\right)$, thus we can simplify the expressions as

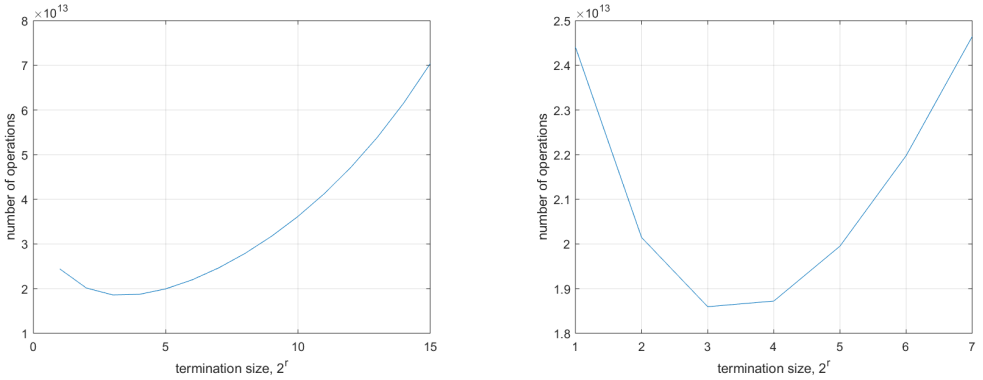
$$7^{(k-r)} \cdot \frac{(2^k)^3}{(2^{(k-r)})^3} = 7^{(k-r)} 8^k 8^{(r-k)} = 7^{k-r} 8^r, \tag{11}$$

and

$$\begin{gathered}\frac{7^{(k-r)}}{(2^{(k-r)})^2}\left(\frac{2^k}{2^{(k-r)}}-1\right)2^{2k}+\frac{18\cdot 4^k}{3}\left(\left(\frac{7}{4}\right)^{(k-r)}-1\right)= \\ 7^{(k-r)}4^{(r-k)}\left(2^k2^{r-k}-1\right)4^k+6\cdot 7^{k-r}4^k4^{r-k}-6\cdot 4^k= \\ 4^r\left(2^r+5\right)7^{(k-r)}-6\cdot 4^k.\end{gathered} \tag{12}$$

The total number of operations that the algorithm requires when the matrix size is 2^k and the termination size is 2^r is thusly

$$\begin{array}{ll} 7^{k-r}8^r & \textit{multiplications} \\ 4^r\left(2^r+5\right)7^{(k-r)}-6\cdot 4^k & \textit{additions.}\end{array} \tag{13}$$



(a) Number of operations while $k = 15$. (b) Number of operations while $k = 7$.

Figure 5: The number of operations performed by the algorithm as a function of the termination criterion 2^r while k is constant.

When analytically searching for the r which minimizes the number of operations of the algorithm it is discovered that $r = 3$ is optimal, regardless of matrix size 2^k (figure 5). Using this r we find that for large k the dominant part of expression will be $3.91 \cdot 7^k$. From this we get

$$3.91\cdot 7^k = 3.91\cdot 2^{\log_2(7^k)} = 3.91\cdot 2^{k\cdot\log_2(7)} = 3.91\cdot\left(2^k\right)^{\log_2(7)}, \tag{14}$$

and because 2^k is the matrix size and the input to the algorithm, denoted m. Now we can determine that

$$\left(2^k\right)^{\log_2(7)} = m^{\log_2(7)} \approx m^{2.801\ldots}. \tag{15}$$

From this, it is clear that the number of floating point operations per second will not grow faster than $\mathcal{O}(m^{\log_2(7)})$ [4, p. 74-77] [5].

3.2 Regression Analysis

When the theoretical complexity of Strassen is tested, and compared with the calculated complexity, it is estimated with a linear regression function. This function

is estimated using that a polynomial function can be rewritten as a linear function by

$$\log y = \log u^k = k \log u. \tag{16}$$

The regression function is the estimated by the linear least square method. We want minimize the residuals of our measured values and our estimated values. We want to find the parameters α_0, α_1 such that the linear function "almost" satisfies $\alpha_0 x + \alpha_1 + \epsilon = y$, where y is a observation of the CPU time[2] for the algorithm. One can also chose to include an error term denoted ϵ which gives an indication of how accurate the estimation is. The linear least square function is determined by solving the following problem [6]

$$\begin{aligned} \min \quad & (\mathbf{A}\ \mathbf{x} - \mathbf{b})^\top (\mathbf{A}\ \mathbf{x} - \mathbf{b}) \\ \text{subject to} \quad & x \in \mathbb{R}^n, \end{aligned} \tag{17}$$

or

$$S = \min \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \alpha_j \varphi_j(t_i) - s_i \right)^2. \tag{18}$$

Here $\mathbf{x}$ is a matrix containing the unknown coefficients, and $\mathbf{b}$ how the computational time depends on the matrix size. $\mathbf{S}$ is a matrix containing feasible coefficients to the regression function. By solving this quadratic optimization problem a linear function, which slope coefficient will be an estimator of the complexity (equation 16), can be estimated. The problem can be solved using the *MultivariateStats* [7] package in *Julia* and allowing us to approximate the complexity in an effective manner.

4 Numerical Stability

4.1 Floating Points

Comparisons were made in this report between different precisions in the implementation of Strassen. Namely half-, single- and double-precision floating points to a reference with extended-precision, using Julia's BigFloat command.

[2]The runtime, computational-, and CPU time in this context are different terms, all referring to the amount of time used to process an algorithm, or other instructions.

Precision	***Sign***	***Exponent***	***Mantissa***	***Tot. bits***	***bytes***
Half, Float16	1	5	10	16	2
Single, Float32	1	8	23	32	4
Double, Float64	1	11	52	64	8
Extended, BigFloat	1	15	64	80	10

Table 1: Differences between the used precisions.

The different precisions imply the number of bits by which each floating point is stored, e.g. a Float32 number is stored with 23 figures in the mantissa (table 1). Floating points are generally stored as the subset F, the system of real numbers of the form

$$y = \pm n \cdot \beta^{e-t} \tag{19}$$

where β is the base, e is the range, t is called the precision, and n, called the significand (or mantissa), is an integer $0 \leq n \leq \beta^t - 1$. The range of the floating numbers is $y\epsilon\{\beta^{e_{min}-1}, \beta^{e_{max}}(1-\beta^{-t})\}$. Floating point numbers are not equally spaced on the real number axis [8].

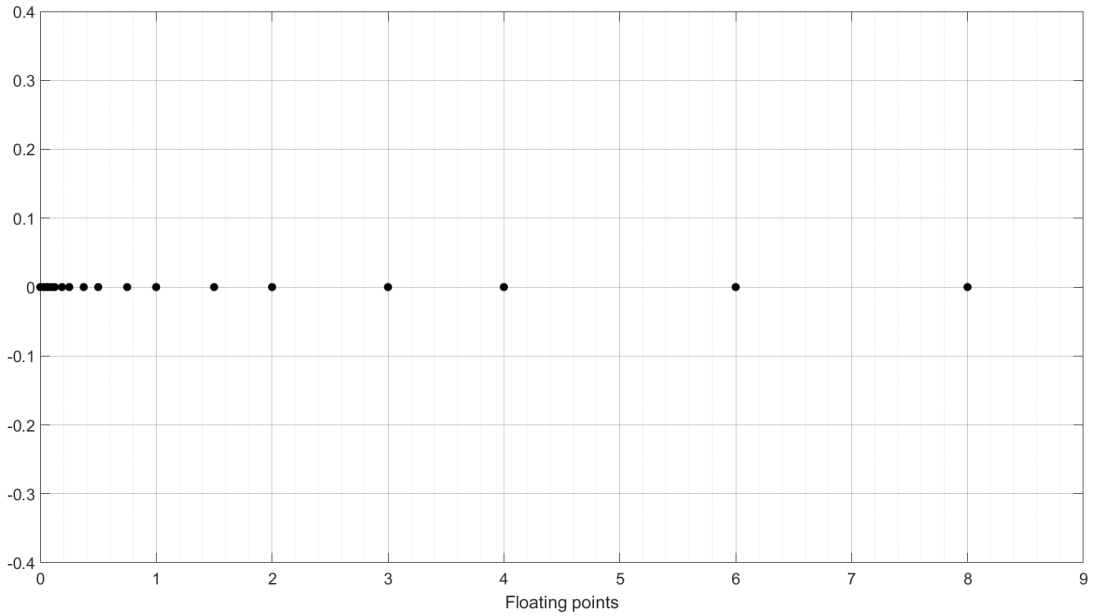


Figure 6: Floating points distribution with a range $0.03125 \leq y \leq 12$ and a unit rounding of smaller than $u = 0.25$.

The spacing between the floating point 1 and the smallest, closest number β is called the machine epsilon, ϵ_M, and is defined $\epsilon_M = \beta^{1-t}$.

The IEEE-standard (Institute of Electrical and Electronic Engineers) requires all rounding arithmetic operations of floating points to be deterministic. When an algebraic operation is executed there are usually a number of floating point representations of the exact result. The IEEE-standard is deterministic in the sense that when performing an operation twice the result will be represented by the same floating point. The example used in [8] demonstrates this as

$$\begin{aligned} a &= \sqrt{\pi} - e \\ b &= \sqrt{\pi} - e \end{aligned} \tag{20}$$

should always be

$$a - b = 0. \tag{21}$$

The rounding errors occur when a number $x \epsilon \mathbb{R}$ is mapped onto a floating point system $F \subset \mathbb{R}$ and represented with a floating point $\tilde{x}$. The errors that can occur when this is done are measure in either the absolute error

$$e_x = |x - \tilde{x}|, \tag{22}$$

or the relative error

$$r_x = \frac{|x - \tilde{x}|}{|x|}. \tag{23}$$

The IEEE-standard guarantees that this relative error is

$$\frac{|x - \tilde{x}|}{|x|} \quad \leq \quad \frac{1}{2}\epsilon_m, \tag{24}$$

or the absolute error

$$|x - \tilde{x}| \quad \leq \quad \frac{1}{2}|x|\epsilon_m. \tag{25}$$

5 Other Algorithms

Today Strassen's algorithm is not the the fastest matrix multiplication algorithm. In 1990 an algorithm called Coppersmith-Winograd was presented which improved the complexity to $\mathcal{O}(m^{2.375477})$, after this there has been smaller improvements to that (figure 7). Though they achieve a smaller complexity they are not superior in practice compared to Strassen, as they only surpass Strassen on very large matrices, which make them impractical on modern hardware. Further research were instead of an algorithm more similar to Strassen's, focusing on reducing the memory allocation.

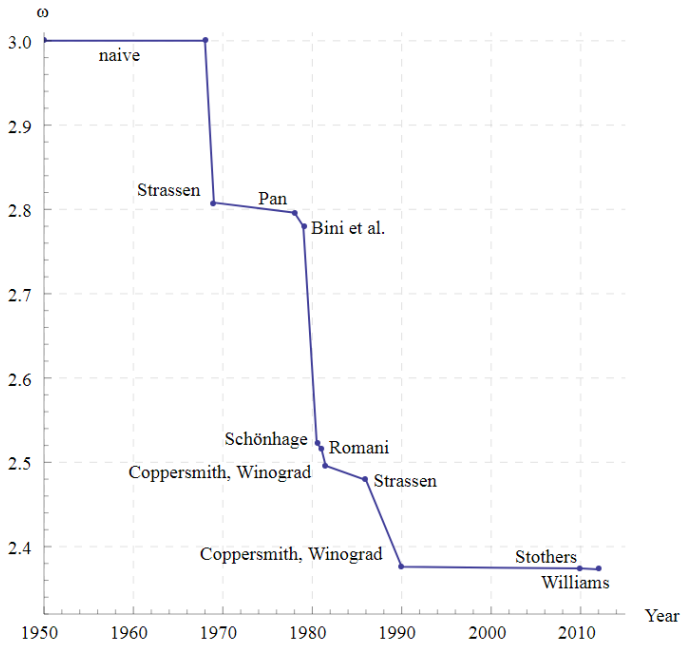


Figure 7: The lowest ω such that matrix multiplication is known to be in $\mathcal{O}(m^\omega)$, plotted against time [9].

5.1 Strassen-Winograd - Overwriting input matrices

The setup is familiar to Strassen when dividing the matrices into sub-matrices, but the computations are made in a different order. The reason for this is to save allocations and memory, thus making it faster. By doing the computations in a certain order it is possible to only use the space of the input matrices and the empty C-matrix [3], depending on how sophisticated the implementation is.
Comparisons, in time (figure 8) and memory used (figure 9), with other algorithms were made with and without termination criterion.

#	*Computation*	*Location*	#	*Computation*	*Location*
1	$S_3 = A_{11} - A_{21}$	C_{11}	12	$S_4 = A_{12} - S_2$	A_{22}
2	$S_1 = A_{21} + A_{22}$	A_{21}	13	$P_6 = S_2T_2$	C_{22}
3	$T_1 = B_{12} - B_{11}$	C_{22}	14	$U_2 = P_1 + P_6$	C_{22}
4	$T_3 = B_{22} - B_{12}$	B_{12}	15	$P_2 = A_{12}B_{21}$	C_{12}
5	$P_7 = S_3T_3$	C_{21}	16	$U_1 = P_1 + P_2$	C_{11}
6	$S_2 = S_1 - A_{11}$	C_{12}	17	$U_4 = U_2 + P_5$	C_{12}
7	$P_1 = A_{11}B_{11}$	C_{11}	18	$U_3 = U_2 + P_7$	C_{22}
8	$T_2 = B_{22} - T_1$	B_{11}	19	$U_6 = U_3 - P_4$	C_{21}
9	$P_5 = S_1T_1$	A_{11}	20	$U_7 = U_3 + P_5$	C_{22}
10	$T_4 = T_2 - B_{21}$	C_{22}	21	$P_3 = S_4B_{22}$	A_{12}
11	$P_4 = A_{22}T_4$	A_{21}	22	$U_5 = U_4 + P_3$	C_{12}

Table 2: Order of computations and their locations.

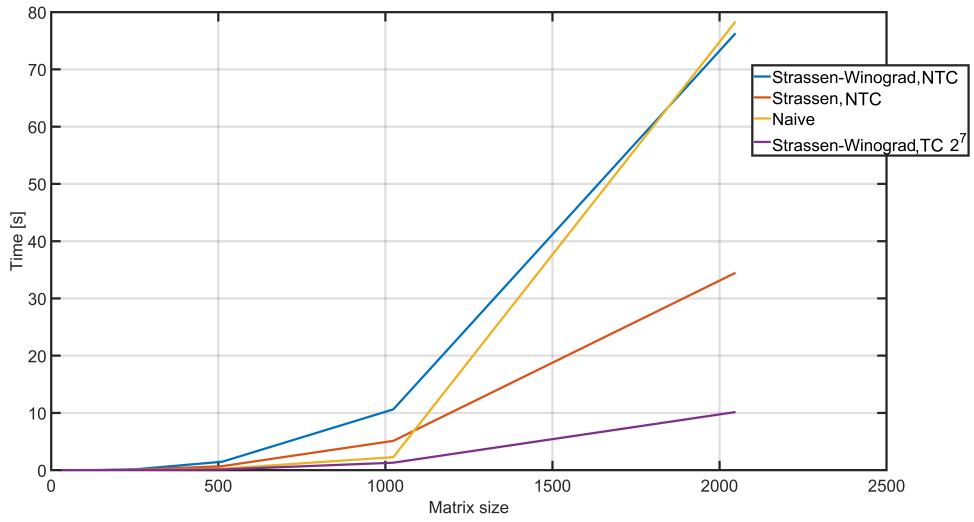


Figure 8: Comparing the median time for different algorithms at different matrix sizes on a i7-6800K, 3.8 GHz, 4×4GB ram.

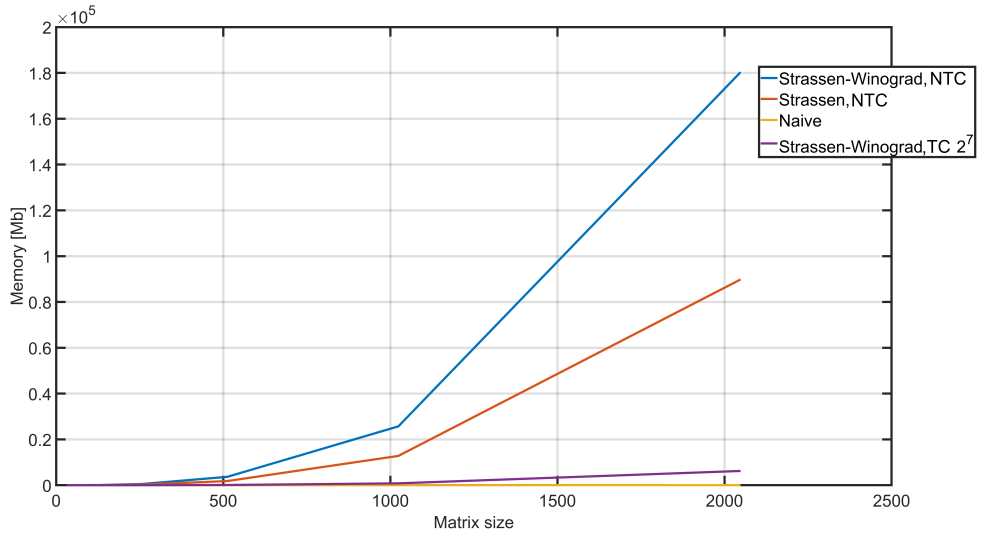


Figure 9: Comparing the memory allocation for different algorithms at different matrix sizes on a i7-6800K, 3.8 GHz, 4×4GB ram.

6 Simulation

All the simulations and computations in this report have been done in the programming language *Julia* [10], as a package in *Atom* [11]. In addition to the main *Julia* software the following packages were added:

- **BenchmarkTools** - This package was used to do the benchmarks of computing times and the memory allocated for them [12].
- **MultivariateStats** - This package was used to do the regression analysis.

"Julia is a high-level, high-performance dynamic programming language for numerical computing. It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library." [10]

6.1 Absolute- and Relative error for different precisions

The input matrices **A** and **B** was randomly generated[3] using half-precision numbers and later converted to either single- or double-precision for calculating the matrix multiplication with Strassen's and the algorithm. The reference result matrix was generated from converting the half-precision matrices **A** and **B** to extended-precision numbers

[3]The half-precision matrices A and B were pseudo-random generated with the command `srand(1)` and `srand(2)` respectively; as shown in algorithm 2. `srand` gives a seed to the random number generator, making it reproducible.

$$\mathbf{A}_{Float16} \rightarrow \mathbf{A}_{Float32}, \mathbf{A}_{Float64}, \mathbf{A}_{BigFloat}$$
$$\mathbf{B}_{Float16} \rightarrow \mathbf{B}_{Float32}, \mathbf{B}_{Float64}, \mathbf{B}_{BigFloat}$$
$$\tilde{\mathbf{C}}_{Float16} = \mathbf{A}_{Float16}\mathbf{B}_{Float16}$$
$$\tilde{\mathbf{C}}_{Float32} = \mathbf{A}_{Float32}\mathbf{B}_{Float32}$$
$$\tilde{\mathbf{C}}_{Float64} = \mathbf{A}_{Float64}\mathbf{B}_{Float64}$$
$$\mathbf{C}_{ref.} = \mathbf{A}_{BigFloat}\mathbf{B}_{BigFloat}$$

Following is an example how the different precision matrices were created.

Algorithm 2 Precisions

```
m = 2^n
srand(1)
A = Matrix{Float16}(randn(m,m))
srand(2)
B = Matrix{Float16}(randn(m,m))
A_b = Matrix{BigFloat}(A)
B_b = Matrix{BigFloat}(B)
C_16 = strassen(A, B)
C_b = naive(A_b, B_b)
e_16 = maximum(abs.(C_b - C_16))
r_16 = maximum(abs.(C_b - C_16)./abs.(C_b))
```

then calculating the matrix multiplication with the two algorithms. With this the absolute (equation 22) and relative error (equation 23) could be calculated for the different precisions, e.g.

$$e_{16} = |C_{ref.} - \tilde{C}_{Float16}|$$
$$r_{16} = |C_{ref.} - \tilde{C}_{Float16}||C_{ref}|^{-1}$$

Comparisons were also made between the different precisions in Strassen to the Naive (figure 10), Strassen to Strassen and Naive to Naive reference matrix, both in terms of absolute error (figure 11) and relative error (figure 12).

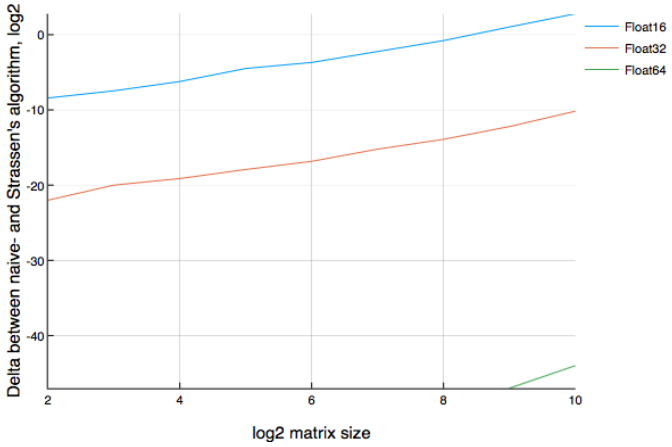


Figure 10: The absoulute error between $\tilde{C}_{strassen}$ and $C_{ref,naive}$.

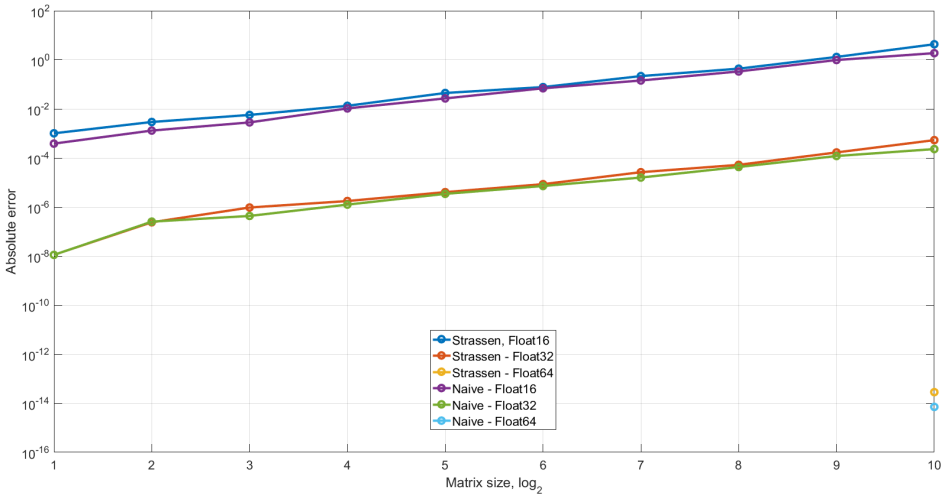


Figure 11: The maximum absolute error for different floating points.

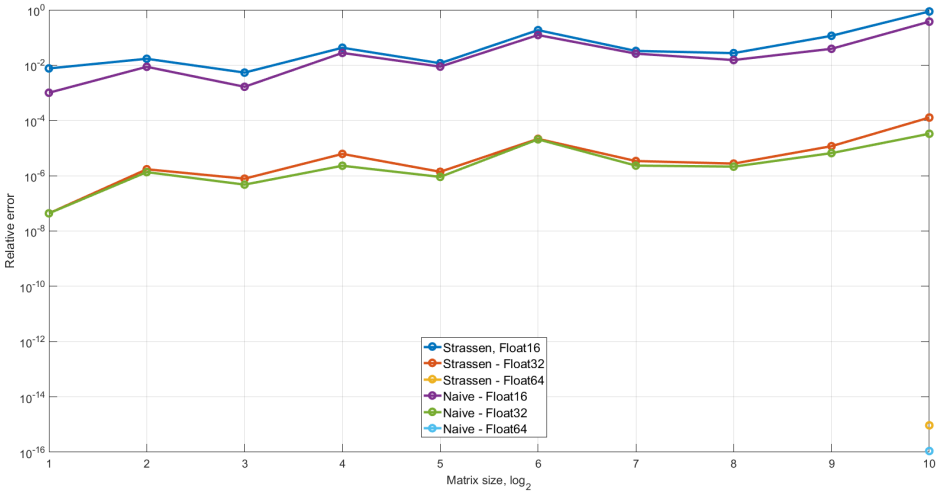


Figure 12: The maximum relative error for different floating points.

6.2 Benchmarking

To confirm the theory of the complexity of the algorithm we created random matrices of size $2^k, k = 2, 3, ..., n$ and ran the function for each matrix size, benchmarking each iteration. Benchmarking in this sens refers to Julia's BenchmarkingTools package, which runs the function multiple times and retrieves a variety om measurement such as median/minimum/maximum runtime, computational time, and memory allocations.

The minimum- and median time values were collected, as they are robust measurements and not sensitive to outliers, and plotted against matrix size. This comparison was made to ensure when Strassen performs in comparison to the Naive, and Julia's built in algorithms. The comparative study of computational time was done strictly using matrices of size $2^k \times 2^k$, otherwise Strassen would have to pad the matrices and the comparison would not be just.

To confirm that the complexity of the algorithm increases in practice as in theory a linear relationship was fitted to the logarithmic time measurements (as a function of logarithmic matrix size). This linear relationship was determined by linear least square regression analysis. The measurements of the complexity of the algorithm varies between each sample run, if the random matrices are re-generated (`srand` not being used), that is. It also varies depending on if only larger matrix sizes are used (e.g. 32×32 and larger) for the regression analysis, as well as the termination criterion.

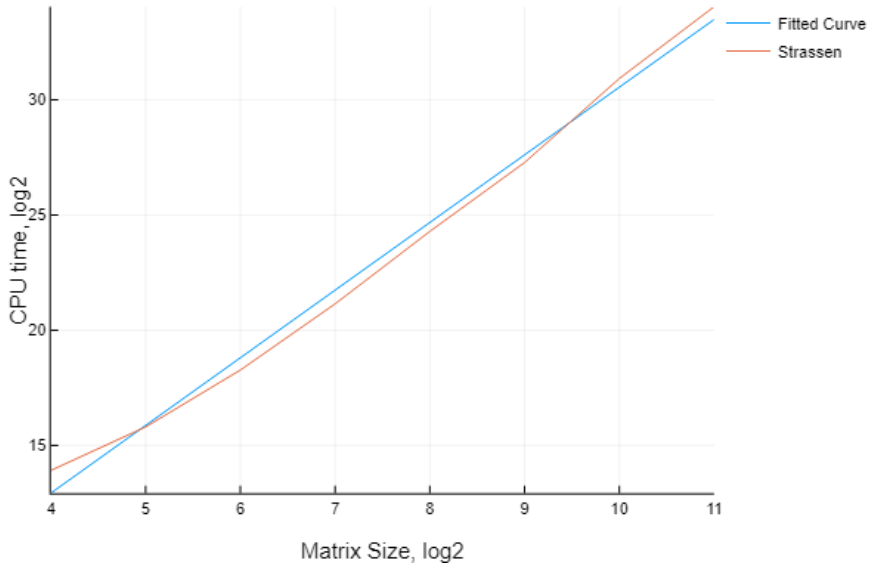


Figure 13: The logarithmic computation time as a function of matrix size, without termination criterion. The fitted curve has a coefficient $k = 2.937$

.

With our estimation of the complexity it is clear that in practice it will not actually be as efficient as $\mathcal{O}(m^{2.807})$. When the algorithm is used with the optimal termination criterion the complexity is only slightly closer to the theoretical value. When run without the termination criterion, i.e. $2^r = 2^2$, the complexity seems to be $\mathcal{O}(m^{2.937})$, (rms-error 0.086), however when run with the termination algorithm $2^r = 2^7$ the complexity is $\mathcal{O}(m^{2.919})$. Compare this with the measured Naive complexity $\mathcal{O}(m^{3.522})$.

Optimizing the basic Strassen by iteratively changing the termination criterion was an straight forward approach to improving the algorithm. The termination criterion for when Strassen should stop being applied recursively, and the block-wise multiplication should be done by the Naive is determined by measuring the calculation time and analyzing where Strassen surpasses the Naive. When analyzing this the smallest possible size for the termination criterion was $2^r = 2^2$, that is, the smallest matrix size the algorithm possibly can be applied to, and can be thought of as Strassen run without termination criterion. The largest size for the termination criterion in theory is $2^r = 2^{k-1}$ i.e. one level Strassen, and the largest practical size for the termination criterion is the matrix size at which Strassen surpasses Naive, when run without termination criterion.

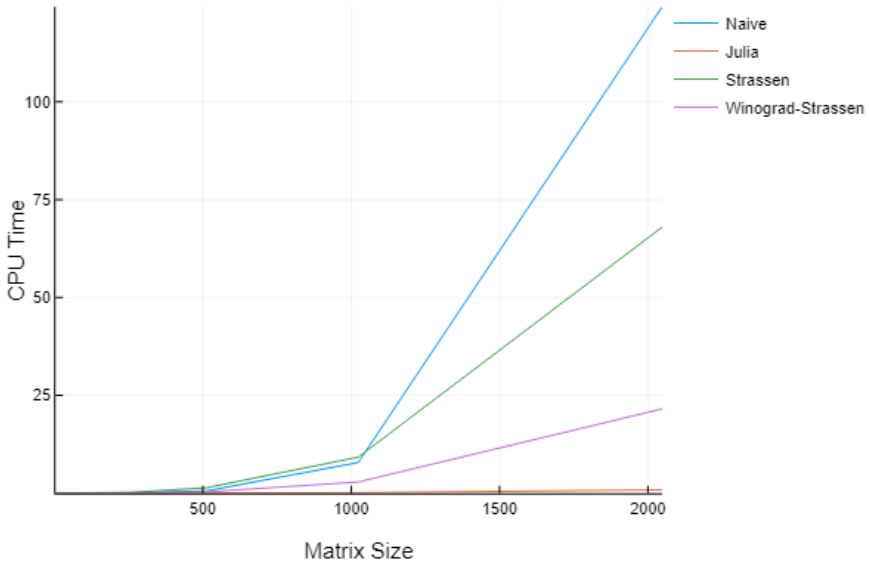
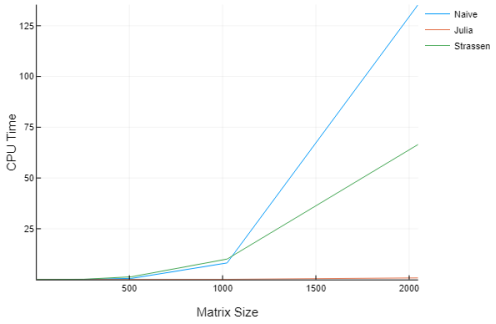


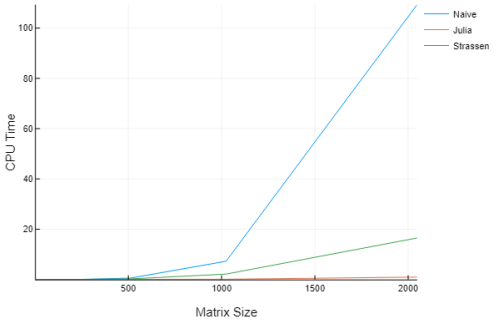
Figure 14: Comparison between the different algorithms, without termination criterion.

In the comparisons our algorithms there seems to be some discrepancies between which matrix size the termination should be at, regarding which time measurement is being used (the minimum- /median-/mean time). However it seems to be clear that Strassen is desirable compared to Naive when the matrix size exceeds 512×512, Strassen's computational time for the 1024×1024 multiplication is notably faster (figure 14). Even faster is Strassen-Winograd which surpasses Strassen with good margins.

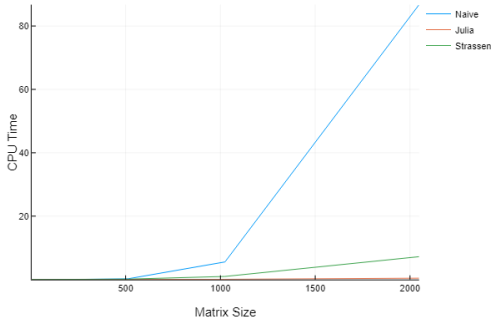
Now we know that the the termination criterion should be $2 \leq r \leq 10$. The optimal value is determined by iteration, comparing CPU time with each r.



(a) Termination criterion $2^r = 2^2$.



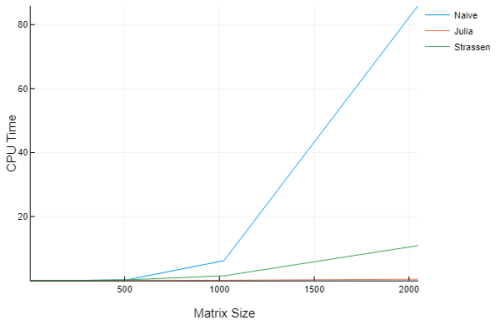
(b) Termination criterion $2^r = 2^5$.



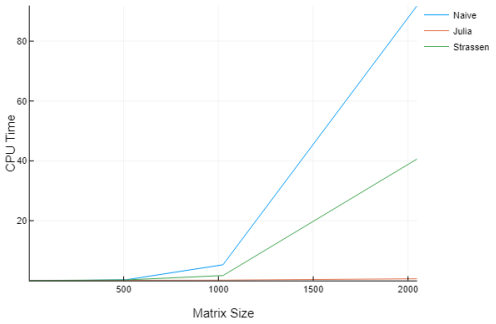
(c) Termination criterion $2^r = 2^7$.

Figure 15: Improved computational time.

The computational time is improved as $r$approaches $r = 7$ (figure 15). As r increases $r > 7$, however, Strassen's runtime starts to increase.



(a) Termination criterion $2^r = 2^8$.



(b) Termination criterion $2^r = 2^{10}$.

Figure 16: Worsened computational time.

It is apparent that the termination criterion will not only optimize the measurements, but also, after a certain size, the algorithm is slowed down (figure 16). The optimal termination criterion seems to be $r = 7$, i.e. size 128×128, which contradicts our theoretically optimal r, size 8×8.

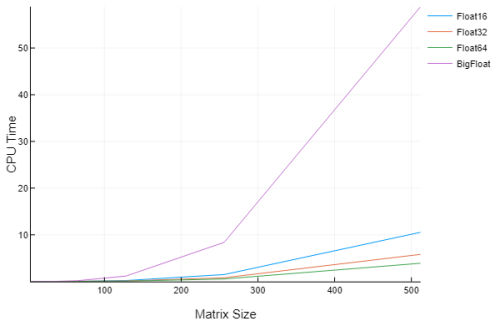
The different algorithms effectiveness is not only measured in terms of computational time, to have a broader basis of evaluation it is relevant to examine how the memory used, or allocated, increases depending on input. That Strassen allocates more memory than the Naive was anticipated as Strassen computes the multiplication in a larger amount of steps.

Matrix size	***Naive***	***Strassen***	***Julia***	***Strassen-Winograd***
16×16	2	3764	1	47808
32×32	2	26402	1	156384
64×64	3	184870	2	586416
128×128	3	1.2942×10^6	2	2.3036×10^6
256×256	3	9.0593×10^6	2	9.1859×10^6
512×512	3	6.3452×10^7	2	9.7333×10^7
1024×1024	3	4.4391×10^8	2	8.1346×10^8
2048×2048	3	3.1073×10^9	2	6.2227×10^9

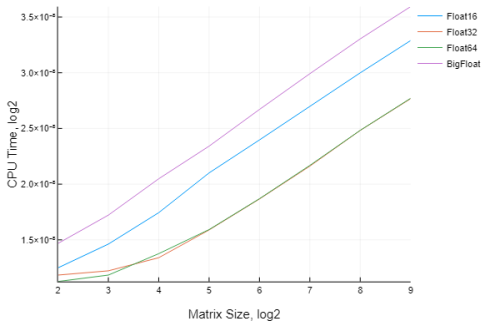
Table 3: Allocations done by the different algorithms when computing the same matrix multiplications.

To exhibit this ineffectiveness of the Strassen and Strassen-Winograd in terms of allocations, they are compared to Julias algorithm and the Naive (table 3). Julias algorithm and the Naive allocates a semi-constant number of allocations, in contrast to Strassen and Strassen-Winograd which has a exponential growth in allocations while they both outperform the Naive in the algorithm runtime.

Along with testing the stability for different floating point precisions, they were also benchmarked to determine if there were any advantages in terms of runtime.



(a) The computational time with Strassen using different float point precisions, $r = 2$.



(b) The computational time with Strassen using different float point precisions, $r = 7$ (logarithmic).

When different floats were benchmarked with termination criterion $r = 2$ the results were opposite to our hypothesis of lower precision implying shorter runtime (figure

17a). They were then re-compared with the optimal termination criterion $r = 7$, which resulted in computational times which were more consistent with our theory: 32-bit precision outdid 64-bit with a slim margin. The 16-bit precision is surprisingly slow (with both termination criteria) when it comes to multiplying matrices, while the other precisions behave according to prediction. Larger precisions entails numbers stored with a larger quantity of numbers to be handled in each operation, as the same algorithm was run for each precision, numbers with Float32 precision should be faster that Float64. The re-compared benchmarked precisions, all with termination criteria $r = 7$, displayed with logarithmic axis (figure 17b) as the differences are too small between 32- and 64-bit precision, and too great between BigFloat and all the other, to be visual otherwise.

7 Discussion

Stability

In order to further describe the requirements of Strassen in implementation one can vary the floating point precisions. As different precisions need different amounts of memory when it comes to allocated space, though they are small separately, the memory will grow exponentially as m gets bigger.

In the case of comparing different precisions with Strassen to the Naive, the results show that the half precision, Float16, is not practical while computing large matrices. Based on our simulations, we conclude that the double precision, Float64, shows to be the more efficient precision. It showed that there were no difference between double- and extended precision until matrix sizes 2^{10}, seen in (figure 11) and (figure 12). Though the decision can be optimized to the specific usage, if the matrix sizes are fixed or what kind of margin of error is accepted.

Implementation

When implementing Strassen one needs to consider how the matrices are shaped. This will have an impact on what the most practical way on how to pad or peel them to a optimal size for recursive computing. For the computations in this report, only power of two-padding was used, in the purpose of following the "pure" Strassen algorithm. When computing large matrices it can make a huge difference between choosing a power of two-padding or a dynamic peeling. For instance if one input matrix of the size of 2049×2049 it would be more practical to peel off one row and column instead of pad it to the next power of two 4096×4096 as it would use more memory than necessary.

In practice the algorithm it can be streamlined making it more useful. The computational algorithms can be set up for parallel computing, dividing the sub-matrix calculations on different CPU-cores.

Complexity

The complexity of Strassen, which was achieved to $\mathcal{O}(m^{2.937})$ compared to $\mathcal{O}(m^{2.807})$, is a rather disappointing result in terms of improved algorithms of matrix multiplication, however the Naive complexity was estimated to $\mathcal{O}(m^{3.522})$, compared with the theoretical $\mathcal{O}(m^3)$. The measured difference between the Strassen and Naive surpasses the theoretical difference which render it a reassuring result. Also there was an improved complexity when Strassen ran with the optimal termination criterion $r = 7$, $\mathcal{O}(m^{2.919})$ compared to $\mathcal{O}(m^{2.937})$. These results indicates that we indeed succeeded in proving that Volker Strassen's algorithm is a superior algorithm to the

Naive not only in theory but in practice.

We found it very interesting that, when analyzing the number of operations executed by the algorithm as a function of r it was clear that $r = 3$ minimized it. In practice it was found that $r = 7$ optimized the computational time while larger r increased it.

Different floating points precisions were also benchmarked to determine their efficiency in addition the their accumulated error. It was found that double-, and single precision performed relatively similar, although single precision outperformed double-, extended precision as presumed. Half precision then ought to be the fastest. As to why this does not entirely hold in practice we do not know, though it might be due to that the algorithm was run on a 64-bit system. We decided not to analyze this further in this report, as it applies more to the subject of computer science.

It should be noted that there are some differences in measurements when benchmarking depending on how "busy" the computer is and what processor they are run on, which also can be a reason for the difference between the optimal theoretical- and practical termination criteria. Moreover is there slight differences depending on the input matrices. These results were made with the same random matrices (using `srand`) to make the results as consistent as possible.

Conclusion

Our objective with this study has been the following. The mathematical operation of computing a matrix-matrix product can be done in different ways. Improving the Naive matrix multiplication for computation of finitely large matrices one is trying to reduce its complexity $\mathcal{O}(m^3)$, with Strassen's algorithm and its theoretical complexity $\mathcal{O}(m^{log_2(7)})$. Comparison of different termination criterion were made and gave the conclusion that, when using $2^k \times 2^k$ for $k = 2, 3, ..., n$, Strassen's algorithm surpasses the Naive algorithm in computation time, [s], when $k \geq 10$ regardless of its termination criterion. Though it was found optimal when applying termination criterion $r = 7$, even though the theoretical termination criterion was at $r = 3$.

While Strassen's algorithm may be faster than the Naive algorithm one needs to consider Strassen's recursive pattern while implementing it, which increases the number of allocations and therefore accumulating bigger rounding errors. If Strassen's is used to constantly allocate new matrices, memory will increase rapidly as m gets bigger. This can be shown with close resemblance Strassen-Winograd's algorithm, which uses a certain computation pattern that over-writes the already existing matrices, therefore saves a lot of memory not allocation any new ones, if implemented correctly.

This paper has confirmed, through implementation in *Julia*, Volker Strassen's proof

from "Gaussian elimination is not optimal" as it is superior to the Naive algorithm for large matrices.

References

[1] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13 (1969), pp. 354–356.

[2] A. Gibiansky. *Matrix Multiplication*. Blog. Mar. 2014. URL: http://andrew.gibiansky.com/blog/mathematics/matrix-multiplication/.

[3] Brice Boyer, Clément Pernet, and Wei Zhou. "Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm". In: *ISSAC 2009—Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*. ACM, New York, 2009, pp. 55–62. DOI: 10.1145/1576702.1576713.

[4] Timothy Sauer. *Numerical Analysis, 2nd ed.* Pearson, 2013.

[5] Xiang Wang. "Analysis of the Time Complexity of Strassen Algorithm". In: *Proceedings of the 2013 International Conference on Software Engineering and Computer Science*. Atlantis Press, 2013. DOI: 10.2991/icsecs-13.2013.21.

[6] Amol Sasane and Krister Svanberg. *Optimization*. Department of Mathematics, Royal Institute of Technology, Stockholm, 2017.

[7] JuliaStats. *MultivariateStats.jl*. Version 0.4.0. [Online; accessed May-2018]. 2018. URL: https://github.com/JuliaStats/MultivariateStats.jl.

[8] E. Jarlebring. *Grundläggande begrepp: Block 0*. [Online; accessed 19-Apr-2018]. Mar. 2018. URL: https://kth.instructure.com/courses/5221/files/851160/download?download_frd=1.

[9] Wikipedia contributors. *Matrix multiplication algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-May-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Matrix_multiplication_algorithm&oldid=829114614.

[10] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (Jan. 2017), pp. 65–98.

[11] Atom. *The hackable text editor*. Version 1.27.1. [Online; accessed Mar-2018]. 2018. URL: https://github.com/atom/atom.

[12] JuliaCI. *BenchmarkTools.jl*. Version 0.3.1. [Online; accessed Apr-2018]. 2018. URL: https://github.com/JuliaCI/BenchmarkTools.jl.

8 Appendix

Tables

Tables concerning measurements of the Naive-, Strassen's-, and Julia's matrix multiplication algorithm

Matrix size	***CPU time [s], Minimum***	***CPU time [s], Median***	***Memory [Mb]***
32×32	1.640×10^{-4}	2.380×10^{-4}	0.673
64×64	1.177×10^{-3}	1.867×10^{-3}	4.995
128×128	9.832×10^{-3}	17.138×10^{-3}	36.081
256×256	81.570×10^{-3}	122.020×10^{-3}	257.025
512×512	652.919×10^{-3}	878.651×10^{-3}	1817.000
1024×1024	5.433	6.074	12.790×10^{3}
2048×2048	38.725	117.831	28.868×10^{4}

Table 4: Measurements concerning the Strassen algorithm, without termination criterion.

Matrix size	***CPU time [s], Minimum***	***CPU time [s], Median***	***Memory [Mb]***
32×32	2.724×10^{-5}	3.701×10^{-4}	0.008
64×64	2.272×10^{-4}	3.228×10^{-4}	0.033
128×128	2.061×10^{-3}	2.977×10^{-3}	0.131
256×256	25.830×10^{-3}	40.035×10^{-3}	0.524
512×512	231.131×10^{-3}	264.893×10^{-3}	2.097
1024×1024	7.872	8.201	8.389
2048×2048	83.521	140.333	3.355

Table 5: Measurements concerning the Naive algorithm.

Matrix size	*CPU time [s], Minimum*	*CPU time [s], Median*	*Memory [Mb]*
32×32	6.169×10^{-6}	1.9534×10^{-5}	0.008
64×64	4.627×10^{-5}	1.7170×10^{-4}	0.033
128×128	1.231×10^{-5}	4.4312×10^{-4}	0.131
256×256	1.064×10^{-4}	0.0019	0.524
512×512	10.627×10^{-3}	0.0167	2.097
1024×1024	74.212×10^{-3}	0.1145	8.389
2048×2048	685.230×10^{-3}	0.8322	3.355

Table 6: Measurements concerning Julia's own algorithm.

Matrix size	*CPU time [s], Minimum*	*CPU time [s], Median*	*Memory [Mb]*
32×32	3.512×10^{-4}	3.686×10^{-4}	1.360×10^{-6}
64×64	2.594×10^{-3}	2.792×10^{-3}	1.008×10^{-5}
128×128	2.460×10^{-2}	2.497×10^{-2}	7.264×10^{-5}
256×256	1.778×10^{-1}	1.803×10^{-1}	5.167×10^{-4}
512×512	1.432	1.482×10^{-3}	0.004
1024×1024	10.630	10.630	0.026
2048×2048	75.549	75.549	0.180

Table 7: Measurements concerning Strassen-Winograd without termination criteria on a i7-6800K, 3.8 GHz, 4×4GB ram.

Matrix size	*CPU time [s], Minimum*	*CPU time [s], Median*	*Memory [Mb]*
32×32	5.150×10^{-5}	5.662×10^{-5}	0.156
64×64	2.756×10^{-4}	2.864×10^{-4}	0.586
128×128	1.944×10^{-3}	2.003×10^{-3}	2.304
256×256	1.405×10^{-2}	1.438×10^{-2}	9.185
512×512	1.118×10^{-1}	1.283×10^{-1}	97.333
1024×1024	1.296	1.311	813.461
2048×2048	10.159	10.159	6.223×10^{3}

Table 8: Measurements concerning Strassen-Winograd with termination criteria 2^7 on a i7-6800K, 3.8 GHz, 4×4GB ram.