# CS124 Programming Assignment 1

Alexander Lin, Jessica Wang

March 25, 2016

## Analytical Analysis

We found the recursive equation of Strassens to be $T(n) = 7T(\lceil \frac{n}{2} \rceil) + 18(\lceil \frac{n}{2} \rceil)^2$. This is because if we are given matrices of size $n$x$n$, Strassen's Algorithm will recurse by doing 7 multiplications on $\lceil \frac{n}{2} \rceil$ size matrices which gives us the recursive element of the equation. Strassen's will additionally do 18 additions of $\lceil \frac{n}{2} \rceil$ matrices which will cost $18 * (\lceil \frac{n}{2} \rceil))^2$. Additionally we know that the conventional matrix mulitplication for matrices of size $n$x$n$ will cost $n^2(2n-1)$ because for each of the $n^2$ elements it will cost $2n-1$ to calculate the value.

To find the analytical cross over point we used python by iterating through cross over points ($n_0$) for various dimensions of matrices ($n$). The following pseudo code describes our algorithm.

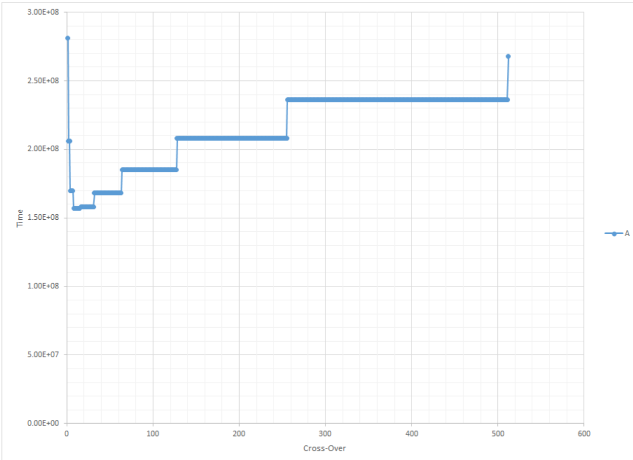
```
function STRASSEN(n, n_0)
    if n ≤ n_0 then
        n^2(2n − 1)
    else if n is even then
        7 ∗ strassen(n/2, n_0) + 18 ∗ (n/2)^2
    else
        7 ∗ strassen(n/2 + 1, n_0) + 18 ∗ (n/2 + 1)^2
    end if
end function
for n_0 ≤ n do
    Return strassen(n_0, n)
end for
```

We ran this pseudocode on various values of $n$. When looking at the values that are returned we want to find the value of $n_0$ that returns the lowest cost. If there are multiple values of $n_0$ that all return the same minimum cost we can represent the crossover point as a range.

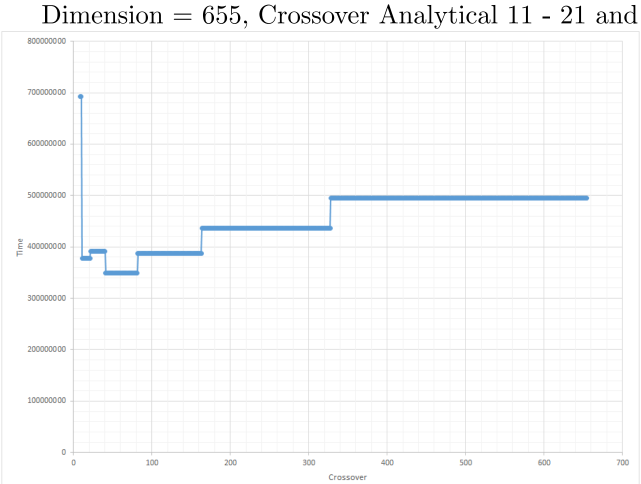
## Analytical Results

The graphs of the crossover numbers can be found below. Exact numbers that are displayed can be found within out code.

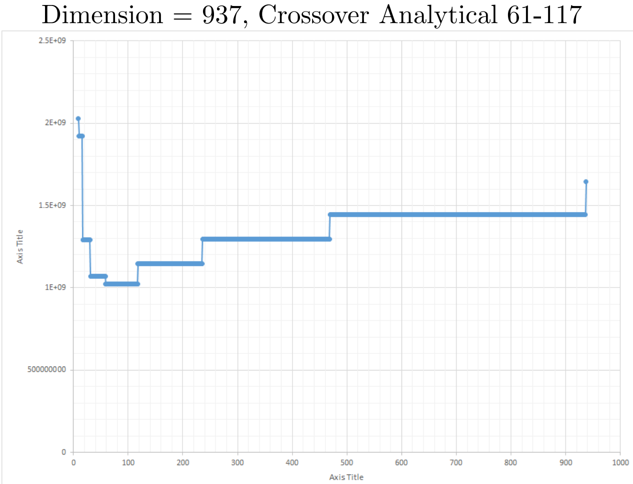
Dimension = 512, Crossover Analytical: 8-15



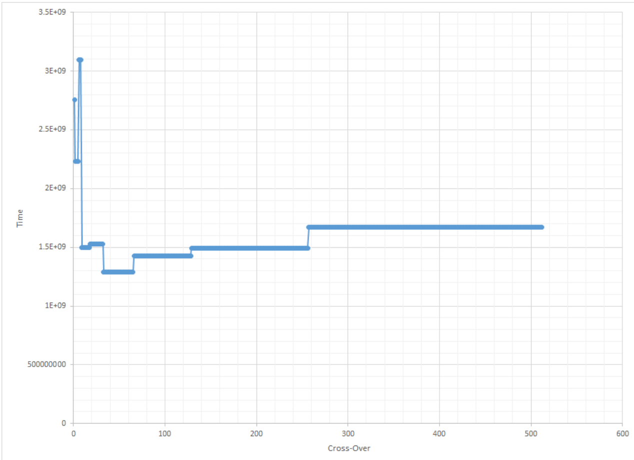

Dimension = 655, Crossover Analytical 11 - 21 and 41-81



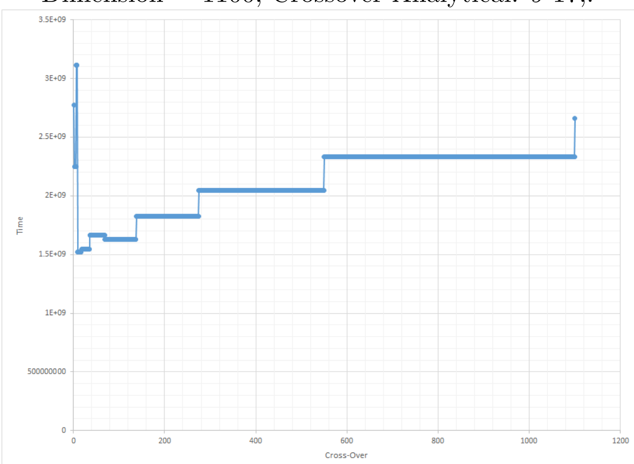

Dimension = 937, Crossover Analytical 61-117



Dimension = 1024, Crossover Analytical: 8-15




Dimension = 1100, Crossover Analytical: 9-17,:




# Analysis of Results

## $2^n$ Values

For dimensions of $2^n$ we find the crossover point to be 16. This is consistent through all the values of $n$ that were powers of 2 that we tested.

## Not $2^n$ Values

For dimensions that are not $2^n$ (particularly dimensions that are odd) there is no clear crossover point that covers all cases. This is most likely because these cases will involve padding the matrix with 0s at some point in the process which will increase the cost of calculating the multiplication. The amount the cost increases would vary with different dimension values. For even dimensions, the crossover point hovers around 16 possibly slightly higher. For odd dimensions we found it to be pretty consistent for $n_0$ to hover above 39.

# Experimental Analysis

We implemented Strassen's Algorithm in C. We stored our matrices as an array of arrays. We decided to do this because it was the most intuitive to us and easier to index into correctly. The array of arrays take an extra step to allocate memory for and thus a little more time, but the memory space will be the same. To do this we created the helper function $addMatrix(matA, ai, aj, matB, bi, bj, dim, sub, returnmat)$ which returned $returnmat$ which added the $dim$x$dim$ size of $matA$ starting at $matA[ai][aj]$ and the $dim$x$dim$ size of $matB$ starting at $matB[bi][bj]$ if $sub = 1$ and subtracted the matrices if $sub = -1$. Helper function $copyMatrix(mat, i, j, dim)$ which will copy the $dim$x$dim$ size of $mat$ starting at $mat[i][j]$ into $returnmat$. The last helper function $combineMatrix(matA, matB, matC, matD, dim)$ will combine four $dim$x$dim$ matrices such that the $returnmat$ will be

$$\begin{pmatrix} matA & matB \\ matC & matD \end{pmatrix}$$

The first step within our recursive Strassen function is to determine whether the dimension of the current matrices attempting to be multiplied is less than or equal to $n_0$. If it is, then we have passed the cross over point and Strassen will revert to conventional multiplication. Otherwise Strassens will recurse and calculate the multiplication using the 7 multiplications of Strassen's algorithm. Strassen's algorithm is calculated by first calculating the both matrices of each multiplication, multiplying those seven pairs together then add the 7 multiplications to get the final matrix.

Our code can be described with the following pseudocode

**function** ADDMATRIX($matA, ai, aj, matB, bi, bj, dim, sub, returnmat$)
  Described above
**end function**
**function** COPYMATRIX($mat, i, j, dim, returnmat$)
  Described above
**end function**
**function** COMBINEMATRIX($matA, matB, matC, matD, dim, returnmat$)
  Described above
**end function**
**function** STRASSEN($matA, matB, dim, n_0, returnmat$)
  **if** $dim \leq n_0$ **then**
    Return $matA * matB$ using conventional multiplication
  **end if**
  $addMatrix(matA, 0, 0, matA, dim/2, dim/2, dim/2, 1, m1a)$;
  $addMatrix(matB, 0, 0, matB, dim/2, dim/2, dim/2, 1, m1b)$;
  $strassen(m1a, m1b, dim/2, n0, m1)$;

  $addMatrix(matA, dim/2, 0, matA, dim/2, dim/2, dim/2, 1, m2a)$;
  $copyMatrix(matB, 0, 0, dim/2, m2b)$;
  $strassen(m2a, m2b, dim/2, n0, m2)$;

  $copyMatrix(matA, 0, 0, dim/2, m3a)$;
  $addMatrix(matB, 0, dim/2, matB, dim/2, dim/2, dim/2, -1, m3b)$;
  $strassen(m3a, m3b, dim/2, n0, m3)$;

  $copyMatrix(matA, dim/2, dim/2, dim/2, m4a)$;
  $addMatrix(matB, dim/2, 0, matB, 0, 0, dim/2, -1, m4b)$;
  $strassen(m4a, m4b, dim/2, n0, m4)$;

  $addMatrix(matA, 0, 0, matA, 0, dim/2, dim/2, 1, m5a)$;
  $copyMatrix(matB, dim/2, dim/2, dim/2, m5b)$;
  $strassen(m5a, m5b, dim/2, n0, m5)$;

  $addMatrix(matA, dim/2, 0, matA, 0, 0, dim/2, -1, m6a)$;
  $addMatrix(matB, 0, 0, matB, 0, dim/2, dim/2, 1, m6b)$;
  $strassen(m6a, m6b, dim/2, n0, m6)$;

$addMatrix(matA, 0, dim/2, matA, dim/2, dim/2, dim/2, -1, m7a);$
$addMatrix(matB, dim/2, 0, matB, dim/2, dim/2, dim/2, 1, m7b);$
$strassen(m7a, m7b, dim/2, n0, m7);$

$addMatrix(m1, 0, 0, m4, 0, 0, dim/2, 1, c00);$
$addMatrix(c00, 0, 0, m5, 0, 0, dim/2, -1, c00);$
$addMatrix(c00, 0, 0, m7, 0, 0, dim/2, 1, c00);$
$addMatrix(m3, 0, 0, m5, 0, 0, dim/2, 1, c01);$
$addMatrix(m2, 0, 0, m4, 0, 0, dim/2, 1, c10);$
$addMatrix(m1, 0, 0, m2, 0, 0, dim/2, -1, c11);$
$addMatrix(c11, 0, 0, m3, 0, 0, dim/2, 1, c11);$
$addMatrix(c11, 0, 0, m6, 0, 0, dim/2, 1, c11);$

$combineMatrix(c00, c01, c10, c11, dim/2, returnmat);$
**end function**

## Allocating Space

The biggest difficulty we faced was space allocation. We originally had copied each the pair of matrices we wanted to multiply into $a_{00}, a_{01}, a_{10}, a_{11}, b_{00}, b_{01}, b_{10}, b_{11}$ such that $matA =$

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$$

and $matB =$

$$\begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

This process took a lot of time to allocate and deallocate. To avoid allocating this we changed our add function, which originally simply added two matrices of equal size, to take in indices value. This allows you to dictate $a_{00}, a_{01}, a_{10}, a_{11}, b_{00}, b_{01}, b_{10}, b_{11}$ when you add them by specifying the start indices of each into the addition function instead of allocating space to store them.

We additionally tried to change our code so we could allocate space for $m1a, m1b, m2a, m2b...., m7a, m7b$ at the beginning to avoid allocating space each time Strassen's occurs by implementing the same indices technique into the *strassen* function. We were able to make it work for 2x2 matrices, but ran into override issues for larger matrices as Strassen's tried to recurse.

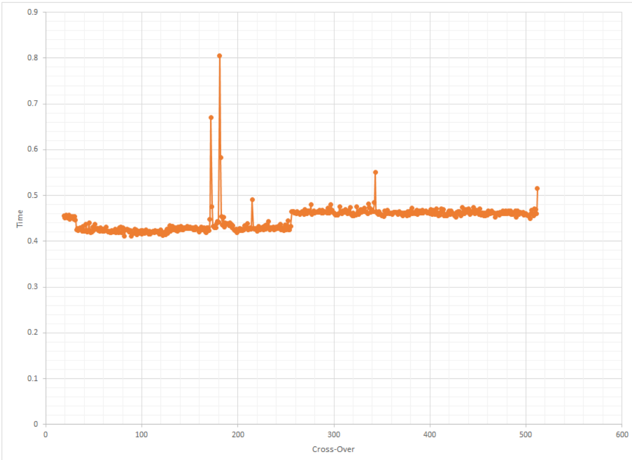
## Dimensions not $2^n$

To deal with odd dimensions we decided to alter the original matrices that would initialized the *strassen* function to be of dimension $2^n$ (the closest power of 2 to the initial dimension) with 0s filling the extra space. We decided to do this before calling the *strassen* function because it allows us to not check the dimension of the matrices within the *strassen* function.

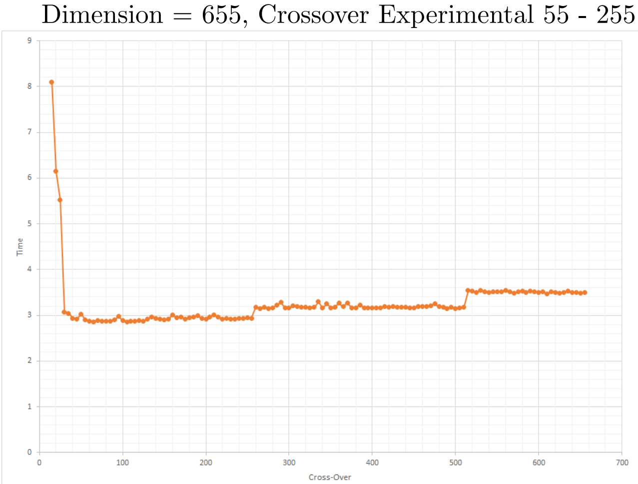
## Experimental Results

We tested experimental results using an external BASH script to time our Strassen implementation. For values that were feasible, we tested $n_0$ with $\Delta = 1$, on larger values we changed $\Delta$ to be some larger value to maximize time efficiency. Attached to our pset we have included a pdf that includes all our experimental datapoints; and below are side by side comparisons of the analytical calculations with experimental calculations.

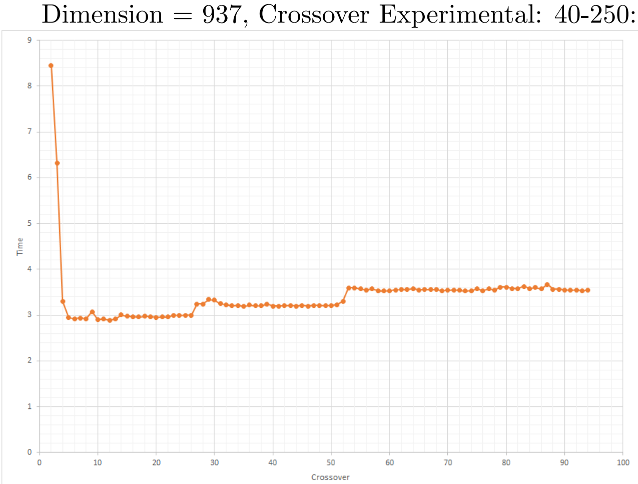
Dimension = 512, Crossover Experimental: 34-225:



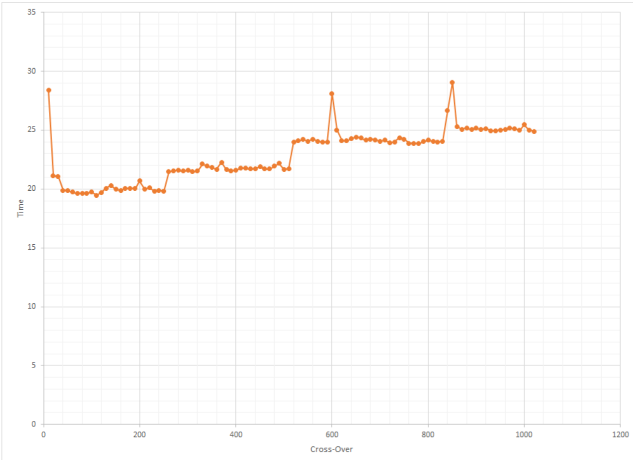

Dimension = 655, Crossover Experimental 55 - 255



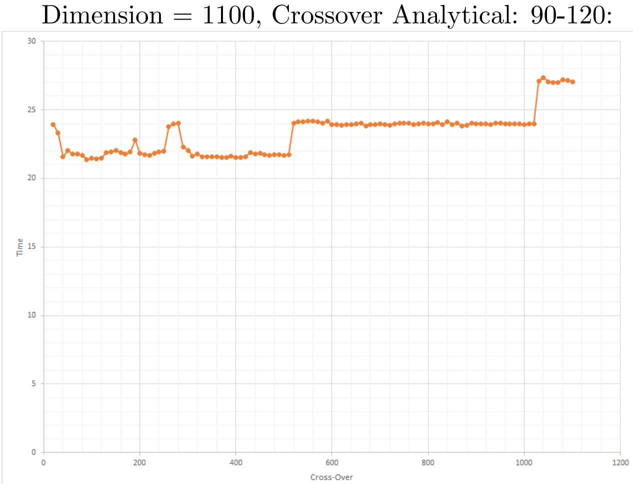

Dimension = 937, Crossover Experimental: 40-250:



Dimension = 1024, Crossover Experimental: 40-110:




Dimension = 1100, Crossover Analytical: 90-120:




## $2^n$ Values

The analytical results indicate that the crossover point should be around 16. The experimental results indicate that this value should be higher, which indicates that Strassen takes a longer time in comparison to the conventional multiplication. This is most likely because the implementation to Strassen not only involves calculating values but also allocating and deallocating space. This takes a good deal of time, which would slow down Strassen's Algorithm in comparison to the conventional algorithm, which does not allocate or deallocate space. Additionally as values of $2^n$ gets larger, the crossover point increases because there will be more memory being allocated and deallocated as $2^n$ grows. Experimental results were harder to define a threshold for $n_0$ since memory allocation shifted results quite a bit; what is certain is that experimental $n_0 >$ analytical $n_0$, however from our observations of dimensions 512 and 1024, it seems $34 < n_0 < 250$ with our best estimations.

## Not $2^n$ Values

The same increase the crossover point in comparison to the analytical result as described for the $2^n$ values is shown in not $2^n$ values. Similar to the $2^n$ values, this has to do with memory allocation and deallocation. The best estimate we can provide given the data that is present is that for not $2^n$ values, $55 < n_0 < 255$