Contents

1 Strassen's Analysis

1.1 Theoretical Calculation of n_0 (cross-over point)

To find n_0, we compared the number of operations it took for each version of matrix multiplication (conventional and Strassen's) to compute the output matrix, s.t. conventional runs normally and Strassen's switches to conventional for matrix operations happening in its subproblems.

For conventional multiplication, each entry of the output matrix takes n multiplications and $n-1$ additions, so a total time of $2n-1$. Since there are n^2 total entries, the total time it takes to calculate the output matrix the conventional way is $n^2(2n-1)$ or $\mathbf{2n^3 - n^2}$.

For Strassen's matrix multiplication, we use with 7 subproblems to multiply two nxn matrices:

$$\begin{aligned} P_1 &= A(F-H) \\ P_2 &= (A+B)H \\ P_3 &= (C+D)E \\ P_4 &= D(G-E) \\ P_5 &= (A+D)(E+H) \\ P_6 &= (B-D)(G+H) \\ P_7 &= (C-A)(E+F) \end{aligned}$$

Where each letter (A through H) represents a matrix quadrant of size $\frac{n}{2}$x$\frac{n}{2}$.

We approach Strassen's runtime as if after splitting each nxn matrix into four $\frac{n}{2}x\frac{n}{2}$ matrices, we switch to conventional matrix multiplication. Furthermore, adding and subtracting two nxn matrices takes a runtime of n^2 (the number of regular number additions/subtractions. So the runtimes of the subproblems are as follows:

$$\begin{aligned} P_{1,2,3,4} &= 2(\frac{n}{2})^3 - (\frac{n}{2})^2 + (\frac{n}{2})^2 \\ P_{5,6,7} &= 2(\frac{n}{2})^3 - (\frac{n}{2})^2 + 2(\frac{n}{2})^2 \end{aligned}$$

So, the total runtime of the seven subproblems is $4(2(\frac{n}{2})^3 - (\frac{n}{2})^2 + (\frac{n}{2})^2) + 3(2(\frac{n}{2})^3 - (\frac{n}{2})^2 + 2(\frac{n}{2})^2)$ or $14(\frac{n}{2})^3 + 3(\frac{n}{2})^2$. To combine these subproblems to make up the output matrix of Strassen's, we use a total of 8 matrix additions/subtractions. This leaves us with a total runtime of $14(\frac{n}{2})^3 + 3(\frac{n}{2})^2 + 8(\frac{n}{2})^2$ or $\mathbf{14(\frac{n}{2})^3 + 11(\frac{n}{2})^2}$.

We set the runtimes of the conventional method and Strassen's method equal to each other and solve for n, which will be our crossover point:

$$2n^3 - n^2 = 14(\frac{n}{2})^3 + 11(\frac{n}{2})^2$$

$$n = 15 \qquad \textit{Used Desmos}$$

So, the crossover point is $n = 15$, but only for *even* values of n since Strassen assumes that it can evenly split the input matrices on each call.

In order to solve for when n is odd, we simply change the $\frac{n}{2}$ in Strassen's to $\frac{n+1}{2}$ in order to account for the added padding:

$$2n^3 - n^2 = 14(\frac{n+1}{2})^3 + 11(\frac{n+1}{2})^2$$

$$n \approx 37.17 \qquad \textit{Used Desmos}$$

Therefore, the cross-over point is $\mathbf{n_0 = 15}$ when n is *even* and $\mathbf{n_0 \approx 37.17}$ when n is *odd* since these are the values when running Strassen's and conventional should take the same amount of time, and therefore we should switch to conventional for values less than or equal to it.

2 Variant of Strassen's and Conventional Algorithm: Experimental Results

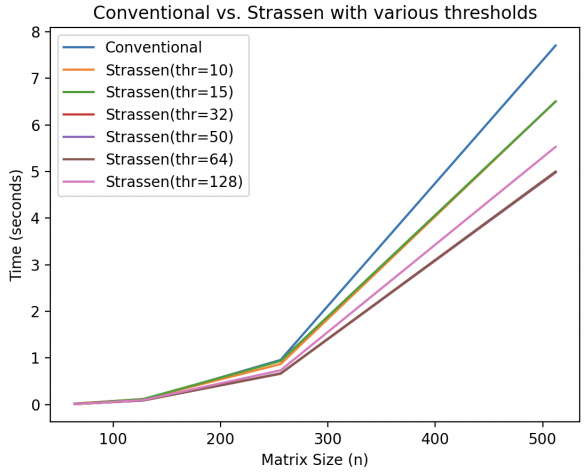
2.1 High Level Implementation

We implemented two matrix-multiplication routines:

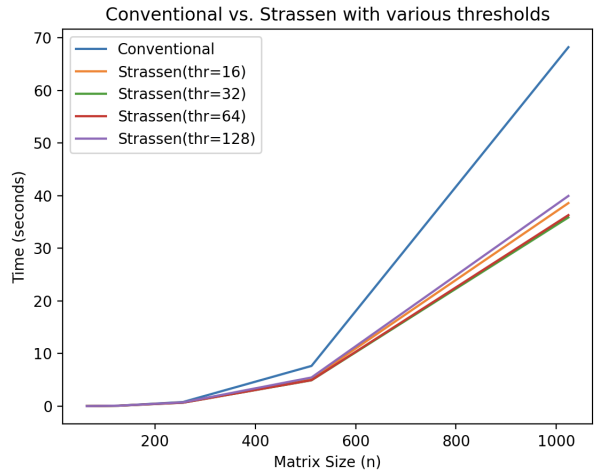
1. Conventional (Naive) $O(n^3)$:
   This uses the triple-nested loop approach: each entry C_{ij} is computed by summing over the products $A_{ik} \cdot B_{kj}$.
2. Strassen's Algorithm:
   - We recursively split the matrices into quadrants, computing 7 subproducts.
   - For subproblems smaller than a threshold `GLOBAL_THRESHOLD`, we switch to the naive $O(n^3)$ routine instead of recursing further.
   - We include "padding" when the matrix dimension is not a power of two.

2.2 Results from Experiments

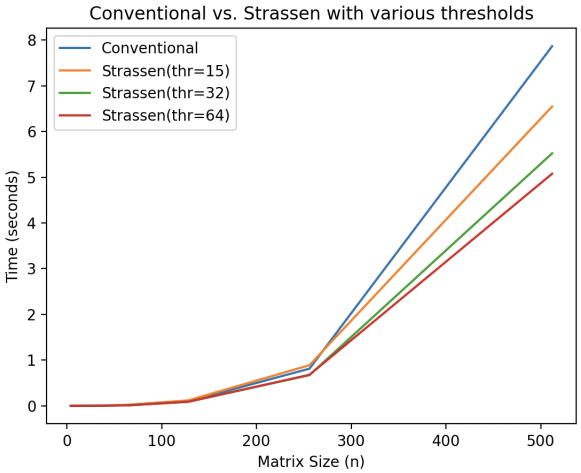
We ran both algorithms on square matrices of sizes $n = 64, 128, 256, 512, 1024$ to see how each threshold reacts to increase dimensions (and in some tests, we experiment with slight alternatives to see a closer look at a window of dimensions as explained later) with integer entries in $\{0, 1, 2\}$. We tested Strassen's with thresholds $\{10, 15, 32, 50, 64, 128\}$ with our theoretical values centered in our list, alongside the conventional method, measuring total execution time in seconds versus dimensions as shown below:



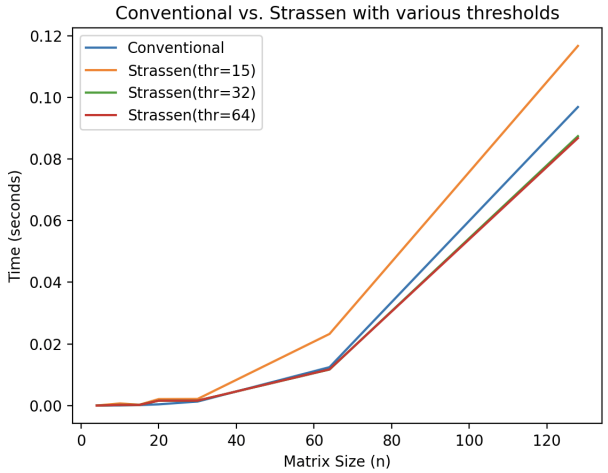
(a) First, we began by graphing the overall performance comparison across all tested matrix sizes. Strassen's method begins to show benefit around mid-size inputs. It is also considerable that certain threshold (such as 10 and 128) may be omitted in future graphs for time and space optimization. Note that we did not graph up to 1000 for time optimization given it is our first graph.



(b) For our second graph we aimed to have a Zoomed-out performance to $n = 1000$. Threshold 32 consistently outperforms others for large n, however for a short while we see threshold 64 begin the best choice in the middle range dimensions.



(c) Next we focus our analysis on lower-dimensional matrices ($n < 500$). Threshold 64 has slight advantage in mid-size range as we predicted, despite 32 being the best threshold in the higher dimensions.

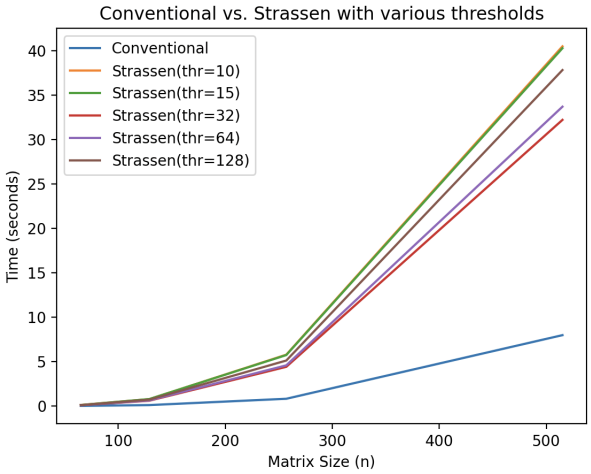


(d) Lastly we see convergence behavior across thresholds. Smaller thresholds introduce overhead, while large ones lose Strassen's benefit, which is in par with our implementation of our variant of Strassen's algorithm.

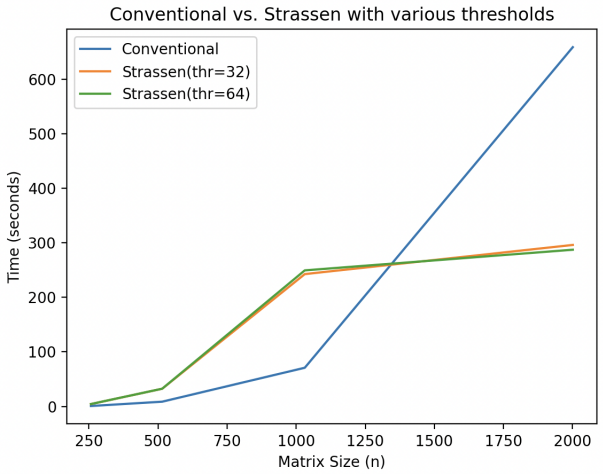
- Experimental Crossover Points (Thresholds).
  From our plots, we observe that:
  - For large n (e.g. 500–1000+), Strassen's with threshold = 32 tends to yield the fastest runtime.
  - For "medium" ranges (roughly 300–500), a threshold of 64 is sometimes marginally faster.
  - Thresholds 10 and 15 generally do not outperform the others in practice, despite the theoretical analysis suggesting a crossover near $n = 15$. In fact, for large n, these small thresholds incur too much recursive overhead and slow down the algorithm.
  - A threshold of 128 also does not perform well in most ranges, presumably because we do not exploit Strassen's speedup until the submatrices are quite large, losing potential gains in many cases.

Therefore, we see that the "optimal" threshold depends somewhat on n itself, but 32 is a consistent winner in our tests, hence why it is chosen in part 3 of our progset.

However, let us consider odd dimensions for our matrices by graphing similar to how we did for our even dimensions:



(a) First, we began by graphing the overall performance comparison across all tested matrix sizes. Strassen's method begins to show worse runtime for all size inputs compared to conventional, which is explained below. Additionally, we did not graph up to 1000 here in order to keep this initial comparison fast and focused.



(b) For our second graph, we took our two most promising values of n from our first graph and ran them up to 1031 and 2001, exluding others due to time constraints. But the conclusions are as expected, eventually strassen runs faster!

From our actual results, it is clear that conventional multiplication consistently outperforms Strassen for all the odd dimensions tested, such as 57, 65, 125, and 257. In pure Python (without using NumPy, which might have significantly improved runtime), Strassen's overhead—especially recursion, repeated matrix slicing, and the many additions and subtractions at each recursive level—dominates when n is only in the low-hundreds. This overhead effectively drowns out the algorithm's theoretical $O(n^{2.81})$ advantage. Because these dimensions are odd, the code must pad the matrices to the next power of two (e.g. 65 padded to 128, 257 padded to 512), which can nearly double the effective size. Each new submatrix allocation, function call, and slice operation in Python adds further time, making the naive $O(n^3)$ approach surprisingly faster within this range. For truly large matrices (in the thousands or more), or in a lower-level language with more efficient memory management, Strassen might eventually outperform the naive method. However, in our actual tests with relatively small, odd dimensions implemented in Python, the cost of padding and recursion clearly outweighed any asymptotic benefit, so conventional multiplication remained the best performer.

2.3 Comparison to Prior Analysis

Our prior theoretical analysis suggested that Strassen's algorithm should outperform the conventional $O(n^3)$ approach for relatively small dimensions, with some estimates predicting a crossover near $n = 15$ or $n = 37$. However, our experimental analysis showed that Strassen only begins to consistently outperform the naive method when n is large, typically in the range of 600 to 1000 or more, and that thresholds of 32 or 64 perform best in practice. For smaller matrix sizes, especially those below $n = 500$, conventional multiplication was often faster. A major source of overhead in this Python implementation comes from frequent memory allocations and copying of data, especially when dealing with odd dimensions that must be padded to the next power of two. Each time we pad a matrix (such as going from 257×257 to 512×512), we allocate a brand-new matrix filled with zeros. Then, during each recursive call to Strassen's,

we allocate seven intermediate matrices ($P_1, \ldots, P_7$) plus four submatrices ($c_1, \ldots, c_4$) for the final combination, each of which must be created (and later combined) via additional loops. On top of that, our split function makes four new submatrices for each recursion level, and the `combine` function merges them again, further increasing the cost of repeated list comprehensions and copy operations in Python. Because these allocations happen at every recursion level, the total overhead can be substantial, especially when the dimension is padded from an odd number up to a much larger power of two. All of this is in addition to the extra matrix additions and subtractions required by Strassen's method itself, which also allocate new lists. In practice, these repeated allocations, slicing, and copying can dominate the runtime at smaller or moderately sized n, overshadowing Strassen's theoretical $O(n^{2.81})$ advantage. Hence, for odd dimensions in the low hundreds (e.g. 257 padded to 512), we observe that conventional multiplication often remains faster, simply because it does not incur the same recursive and memory-allocation overheads.

In my implementation, I optimized both the naive and Strassen's approaches to reduce memory allocation overhead. For the naive algorithm, the result matrix is preallocated once in the function `conventional_multiply`, which avoids the need to repeatedly allocate memory during the accumulation of each entry, this is a key optimization that minimizes per-iteration overhead. In the Strassen implementation, although the algorithm inherently requires creating new matrices at several stages (such as when splitting the matrix into four submatrices or computing the seven intermediate products), I mitigated some of the memory overhead by implementing the `add_matrix` and `sub_matrix` functions using Python's list comprehensions. These functions efficiently compute element-wise sums and differences by constructing new matrices in a single, optimized operation, rather than resorting to more verbose and memory-intensive loops.

3 Counting Triangles

3.1 Results vs. Expectation

p	Reported Number of Triangles	Expected Number	Difference (|Reported - Expected|)	% Off (Diff./Expected)
0.01	172.4	178.43	6.03	≈0.034
0.02	1397.4	1427.46	30.06	≈0.021
0.03	4735.6	4817.69	82.09	≈0.017
0.04	11447.4	11419.71	27.69	≈0.0024
0.05	22455.6	22304.13	151.47	≈0.0068

3.2 Analysis and Explanation of Differences

The reported number of triangles for each probability p differs slightly from the expected number, with the greatest difference being only about 3% off from the expected number. We noticed that the difference between the expected number and the reported number generally shrunk as the expected number of triangles increased.

Overall, the reported numbers systematically differ only slightly from the expected numbers. This makes sense since probability doesn't equate to a consistent output, so it would be quite alarming if the reported number of triangles was exactly equal to the expected number. As for the shrinking difference (percentage-wise), it is likely that when given more edges to traverse, the overall deviation from the expected number gets smaller compared to when there are less edges to travel, leading to a decrease in how off from the expected number the reported number is.