Investigation of Efficient Methods for the Determination of Strassen-type Algorithms for Fast Matrix Multiplication

By
Zachary A. MacDonald

A Thesis Submitted to
Saint Mary's University, Halifax, Nova Scotia
in Partial Fulfillment of the Requirements for
the Degree of Bachelor of Science, Honours Computer Science

April, 2016, Halifax, Nova Scotia

Approved: Dr. Paul Muir
Supervisor

Approved: Dr. Gordon MacDonald
Supervisor

Approved: Dr. Stavros Konstantinidis
Reader

Date: April 25, 2016

Investigation of Efficient Methods for the Determination of Strassen-type Algorithms for Fast Matrix Multiplication

by Zachary A. MacDonald

Abstract

Fast matrix multiplication algorithms such as the Strassen algorithm allow for the multiplication of matrices with fewer multiplications than would normally be needed, thus making the calculations more efficient. We introduce an efficient and highly parallelizable method for searching for Strassen-type fast matrix multiplication algorithms for multiplying two 2 x 2 matrices, and discuss its application in searching for algorithms that perform fast matrix multiplies for the 3 x 3 case. Searching for these algorithms is difficult, because the search space of possible algorithms is extremely large. The method introduced in this thesis, for the 2 x 2 case, makes use of a grid representation for the matrix multiplication algorithms, making it easier to view patterns in the algorithms and prune the search space effectively. The method is implemented in the Scilab language and is able to find all Strassen-type fast matrix multiplication algorithms in about 90 minutes of computing time on an Intel Core i7 6700k CPU. Based on the insight we gain from the 2 x 2 case, we provide suggestions for an efficient search method for the 3 x 3 case.

April 25, 2016

1. Introduction

Matrix multiplication is used in a wide variety of computing applications, including weather prediction, flight controls, and graphical computations, but for large matrices it can be a very costly operation.

The standard algorithm requires n^3 multiplications for two square matrices of size n. For several decades going back to the 1960s, there has been interest in attempting to break this n^3 barrier for matrix multiplication. The first breakthrough came in 1969, with the Strassen algorithm. Strassen's algorithm uses approximately $n^{2.805}$ multiplications for square matrices of size n [1]. The Strassen algorithm is based on multiplying two 2×2 matrices using seven multiplications instead of eight, but it scales up to any matrix of size $n \times n$, where $n = 2^m$ for some positive integer m, and matrices of other sizes can be padded with zeros to fit this size requirement. When scaled up to larger matrices in this way, it is applied recursively to matrices of size $2^{m-1} \times 2^{m-1}$ and eliminates one matrix multiplication of a $2^{m-1} \times 2^{m-1}$ matrix. We give a detailed description of Strassen's algorithm in Chapter 3 of the thesis.

For larger matrices, matrix multiplications become increasingly more expensive than matrix additions. For a square matrix of size 2, a matrix addition involves 4 scalar additions, while a matrix multiplication involves 4 scalar additions and 8 scalar multiplications; for a square matrix of size 8, a matrix addition involves 64 scalar additions and a matrix multiplication involves 448 additions and 512 multiplications. A fast matrix multiplication algorithm will involve many more additions than the standard algorithm for a slight reduction in the number of multiplications. Strassen's algorithm involves 18 addition/subtraction operations, 14 more than the standard algorithm.

It is possible that a 3×3-based algorithm could be more efficient than applying Strassen's algorithm to the 3×3 matrices with a padding of a row and column of zeros, but there is no efficient way of finding such an algorithm because the search space of possible algorithms is extremely large. This thesis describes a method that finds all 2×2 Strassen-type algorithms very quickly and efficiently (Ninety minutes with the current implementation in a scripting language, which is known to be between ten and a hundred times slower than a compiled language). This is much faster than any previously known method, and can be logically scaled toward finding algorithms for fast matrix multiplication of 3×3 matrices. It can also benefit greatly from parallelization.

Currently, the best 3×3 multiplication algorithm requires 23 multiplications, or $n^{2.854}$ for a matrix of size n (standard matrix multiplication for 3×3 matrices requires 27 multiplications). This is less efficient than Strassen's algorithm, but it has not been proven that this is optimal, and it is conjectured that there could exist a 3×3 fast matrix multiplication algorithm that uses as few as 19 multiplications [8], or $n^{\log_3(19)} \approx n^{2.68}$ multiplications for a matrix of size $n \times n$, where $n = 3^m$

for some positive integer m. Though the size of the matrix is more restricted when using a 3×3 algorithm, other sizes of matrices can be padded with zeros as in the case of the Strassen algorithm, and this would still be an improvement for a large range of matrix sizes.

In the next chapter, we review the background literature on fast matrix multiplication. In Chapter 3, we detail the mathematical principles of our method of discovering fast matrix multiplication algorithms and introduce the idea of hierachical grids as a framework for the discovery process. In Chapter 4, we extend the mathematical ideas and discuss the implementation of our method and the details of how to create the grids and solution rectangles that determine the fast matrix multiplication algorithms. In the final chapter we summarize this work and discuss the implications of using our method for the discovery of algorithms for fast multiplication of 3×3 matrices.

2. Background Literature on Fast Matrix Multiplication

In 1969, German mathematician Volker Strassen was attempting to prove that the standard algorithm for matrix multiplication was optimal in terms of multiplications, and that 2×2 matrix multiplication required 8 multiplications. By accident, he instead proved that it could be done with 7 multiplications, and discovered an algorithm that accomplished this, now known as Strassen's Algorithm [1].

With Strassen's algorithm, or similar 7-multiplication algorithms, the time complexity of a matrix multiplication is reduced from n^3 multiplications to $n^{\log_2(7)} \approx n^{2.807}$ for a square matrix of size n. However, this is a theoretical time complexity. If a matrix's size is not a power of two, it must be "padded" with zeros for Strassen's algorithm to work, which may impede the performance. When applied to matrices of size $2^m \times 2^m$, Strassen's algorithm prescribes a matrix multiplication in terms of the multiplication of 7 matrices of size $2^{m-1} \times 2^{m-1}$. The original matrices are viewed as each being made up of four $2^{m-1} \times 2^{m-1}$ blocks.

For arbitrarily large matrices, there are other algorithms that are more efficient, in the theoretical sense. Currently the fastest known is the Le Gall algorithm [5], which is based on the earlier Coppersmith-Winograd algorithm [3], and can multiply a pair of $n \times n$ matrices in $O(n^{2.3728639})$ time. This algorithm and others in its family are not used in practice, as they are only optimal on matrices too large to be handled by modern computers.

Due to the increased number of additions in Strassen's algorithm, it does not become more cost-effective until matrices of size 100×100 or greater, but this is still a much smaller value of n than that required to make other types of fast algorithms practical. A scalar addition is approximately as costly as a scalar multiplication on a modern computer, but, as mentioned earlier, a matrix multiplication is far more expensive than a matrix addition, so for larger matrices the cost of the additions can

be safely ignored. Generally, only the standard algorithm and Strassen-type algorithms are used in practice, depending on the size of the matrices. Concerns such as cache usage, space complexity, and numerical instability can make Strassen's algorithm less appealing in certain circumstances, but it has been found to be more efficient on dense matrices than the standard algorithm [6].

It is difficult to discover a Strassen-type algorithm. As we will see in the next chapter, the Strassen algorithm involves the computation of seven intermediate quantities, seemingly based on random expressions involving the original matrix elements. So far, methods of computationally searching for these Strassen-type fast matrix multiplication algorithms have been very costly. In 2010, Oh and Moon [2] did a genetic search for Strassen-type algorithms, and calculated that it would take a Pentium IV 2.4 GHz machine 67 million years to perform an exhaustive search of this type.

MacAdam and MacDonald [7] demonstrated an exhaustive search that ran in 18 hours in C, by taking advantage of equivalence, symmetry, parallelism, and linear algebraic reductions to improve the search. Our method improves on this time, running in 90 minutes in a slower, scripting language, using a home desktop.

In the 3×3 case, the best algorithm currently known was discovered by Julian Laderman in 1976 [4]. It uses 23 multiplications, an improvement over the 27 multiplications of the standard algorithm. Unlike for 2×2 algorithms, however, it has not been proven to be optimal. It is conjectured that the optimal algorithm could use as few as 19 multiplications [8].

3. A method for the discovery of Strassen-type matrix multiplication algorithms

3.1. **Representation of the algorithm.** We begin by reviewing the classic representation of Strassen's algorithm [1]. The multiplication of two 2×2 matrices, $AB = C$, using Strassen's algorithm involves seven intermediate terms, each one being formed with a single matrix multiplication and one or two matrix additions/subtractions:

$$
\begin{aligned}
m_1 &= (a_{11} + a_{22}) \times (b_{11} + b_{22}), \\
m_2 &= (a_{21} + a_{22}) \times b_{11}, \\
m_3 &= a_{11} \times (b_{12} - b_{22}), \\
m_4 &= a_{22} \times (b_{21} - b_{11}), \\
m_5 &= (a_{11} + a_{12}) \times b_{22}, \\
m_6 &= (a_{21} - a_{11}) \times (b_{11} + b_{12}), \\
m_7 &= (a_{12} - a_{22}) \times (b_{21} + b_{22}).
\end{aligned} \tag{1}
$$

Then we calculate each of the four values in the product matrix C as a linear combination of these M-values, using only 1, -1, and 0 as coefficients. If we were to substitute the M-values and simplify these equations, we would see that they will give the same results as the standard algorithm:

$$\begin{aligned} c_{11} &= m_1 + m_4 - m_5 + m_7, \\ c_{12} &= m_3 + m_5, \\ c_{21} &= m_2 + m_4, \\ c_{22} &= m_1 - m_2 + m_3 + m_6. \end{aligned} \tag{2}$$

There are three other families of Strassen-type algorithms that use seven multiplications [2], and there are no similar algorithms that can do this with fewer than seven matrix multiplications [9].

In this chapter, we want to discuss an efficient method for finding these Strassen-type algorithms. The process for discovering Strassen-type algorithms involves representing the M-values and the C-values in terms of certain vector and matrix-vector products.

For a given pair of 2×2 matrices A and B, we create the vector $\mathbf{a}$ by stacking the columns of A and the vector $\mathbf{b}$ by stacking the rows of B, so

$$\mathbf{a} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{12} \\ a_{22} \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_{11} \\ b_{12} \\ b_{21} \\ b_{22} \end{bmatrix}. \tag{3}$$

Then, each of the seven M-values (1) is written as follows. In this form, we can see why each must contain only one multiplication.

$$m_1 = \mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \mathbf{b}, \qquad m_2 = \mathbf{a}^T \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \mathbf{b} \tag{4}$$

$$m_3 = \mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & -1 \end{bmatrix} \mathbf{b}, \qquad m_4 = \mathbf{a}^T \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 & 0 \end{bmatrix} \mathbf{b}$$

$$m_5 = \mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{b}, \qquad m_6 = \mathbf{a}^T \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix} \mathbf{b}$$

$$m_7 = \mathbf{a}^T \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{b}.$$

Similarly, an examination of the C-values based on standard matrix multiplication, e.g.:

$$c_{11} = \begin{bmatrix} a_{11} & a_{12} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = a_{11}b_{11} + a_{12}b_{21},$$

shows that we can write the C-values in terms of $\mathbf{a}$ and $\mathbf{b}$ and four 4×4 matrices as follows:

$$c_{11} = \mathbf{a}^T \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \mathbf{b}, \qquad c_{12} = \mathbf{a}^T \left[\begin{array}{cc|cc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right] \mathbf{b}$$

$$c_{21} = \mathbf{a}^T \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] \mathbf{b}, \qquad c_{22} = \mathbf{a}^T \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \mathbf{b}.$$

When we take the linear combinations of the M-matrices as above, we can see that they give us the same C-matrices, showing that Strassen's algorithm gives the same results as the standard algorithm.

$c_{11} = m_1 + m_4 - m_5 + m_7$:

$$\mathbf{a}^T \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{b} = \mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \mathbf{b} + \mathbf{a}^T \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 & 0 \end{bmatrix} \mathbf{b}$$

$$-\mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{b} + \mathbf{a}^T \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{b}$$

$$\mathbf{a}^T \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{b} = \mathbf{a}^T \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \mathbf{b} + \mathbf{a}^T \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{bmatrix} \mathbf{b}$$

$$-\mathbf{a}^T \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{b} + \mathbf{a}^T \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & -1 \end{bmatrix} \mathbf{b}$$

Similarly,

$c_{12} = m_3 + m_5$:

$$\mathbf{a}^T \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{b} = \mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & -1 \end{bmatrix} \mathbf{b} + \mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{b}.$$

$c_{21} = m_2 + m_4$:

$$\mathbf{a}^T \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{b} = \mathbf{a}^T \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \mathbf{b} + \mathbf{a}^T \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 & 0 \end{bmatrix} \mathbf{b}.$$

$c_{22} = m_1 - m_2 + m_3 + m_6$:

$$\mathbf{a}^T \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{b} = \mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \mathbf{b} - \mathbf{a}^T \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \mathbf{b}$$

$$+\mathbf{a}^T \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & -1 \end{bmatrix} \mathbf{b} + \mathbf{a}^T \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix} \mathbf{b}.$$

The above equations must be true for every possible value of $\mathbf{a}$ and $\mathbf{b}$, so we can remove these vectors from the equations. That is, the Strassen algorithm is based on the fact that

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 & 0 \end{bmatrix}$$

$$- \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix},$$

and corresponding matrix-vector expressions associated with c_{12}, c_{21}, and c_{22}. If we assign names to the vectors appearing in the right hand sides of the above equations, we can rewrite the four equations above as a single summation and define a fast matrix multiplication algorithm in a very compact format. Let

$$\text{(5)} \qquad \zeta_{\mathbf{1}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \zeta_{\mathbf{2}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \zeta_{\mathbf{3}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \zeta_{\mathbf{4}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \zeta_{\mathbf{5}} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \zeta_{\mathbf{6}} = \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \zeta_{\mathbf{7}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix},$$

and

$$(6) \qquad \eta_{\mathbf{1}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \eta_{\mathbf{2}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \eta_{\mathbf{3}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ -1 \end{bmatrix}, \eta_{\mathbf{4}} = \begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \eta_{\mathbf{5}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \eta_{\mathbf{6}} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \eta_{\mathbf{7}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}.$$

(We note that some of the η_i vectors are the same as some of the ζ_i vectors, but some are different). With these definitions, the following equation defines a fast matrix multiplication algorithm for 2×2 matrices, where a, b, c, and d are alternately equal to one with the other three equal to zero:

$$(7) \qquad \begin{bmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix} = \sum_{k=1}^{7} x_k \zeta_k \eta_k^T,$$

where x_k are constants from the set $\{0, 1, -1\}$.

For example, for $b = 1$ and $a = c = d = 0$, the right hand side of (7) is $\zeta_3 \eta_3^T + \zeta_5 \eta_5^T$. $x_3 = x_5 = 1$ and all other x_k values are zero. This corresponds to the equation $c_{12} = m_3 + m_5$ in Strassen's algorithm.

Equation (7) implies that in order to obtain a Strassen-type algorithm, we must be able to find non-zero vectors $\zeta_{\mathbf{k}}$ and $\eta_{\mathbf{k}}$, $k = 1$ to 7, whose entries are $\in \{-1, 0, 1\}$, and four sets of coefficients $x_k \in \{-1, 0, 1\}$ so that the summations add up to give the four target matrices

$$(8) \qquad \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{cc|cc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] \text{ and } \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right].$$

The only requirement for a Strassen-type algorithm is that the algorithm, defined by its ζ_k, η_k, and x_k values, fulfills equation (7). There are four families of such equations, with many small variations (usually involving negative numbers), defined by their arrangement of four-term solutions and two-term solutions. That is, which of their summations have x_k with two non-zero values, and which have x_k with four non-zero values. There are no solutions with any number of terms other than two or four [2].

3.2. **Creating a search method.** The process above is described and used by MacAdam and MacDonald [7]. In their research, it took a compiled program 18 hours to perform an exhaustive search and determine every 2×2 fast matrix multiplication algorithm. By solving this problem in a way that prunes a large number of cases, we will in this thesis develop a method that allows a much slower scripted language to complete an exhaustive search in about an hour and a half. We begin by splitting the ζ_j and η_j vectors into smaller vectors of size 2:

$$\zeta = \left[\begin{array}{c} \mathbf{u} \\ \hline \mathbf{w} \end{array}\right], \eta = \left[\begin{array}{c} \mathbf{r} \\ \hline \mathbf{s} \end{array}\right].$$

There were previously $3^4 = 81$ possible ζ and η vectors (each vector has 4 entries, and there are three ways to choose each entry, -1, 0, 1.) There are now only 9 different vectors, one of which is the zero vector, and four of which are negatives of the other four. We assign each of these non-negative vectors a name for future reference.

$$v_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, v_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, v_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, v_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, v_5 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{9}$$

Then we can split the summation appearing on the right hand side of Equation (7) into four pieces to be solved separately, where $\mathbf{u_k}, \mathbf{w_k}, \mathbf{r_k}, \mathbf{s_k}$ are the decomposed ζ and η vectors, each comprising a set of vectors from v. This gives

$$\left[\begin{array}{c|c} E_{ij} = \sum_{k=1}^{7} x_k \mathbf{u_k} \mathbf{r_k}^T & \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} = \sum_{k=1}^{7} x_k \mathbf{u_k} \mathbf{s_k}^T \\ \hline \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} = \sum_{k=1}^{7} x_k \mathbf{w_k} \mathbf{r_k}^T & E_{ij} = \sum_{k=1}^{7} x_k \mathbf{w_k} \mathbf{s_k}^T \end{array}\right] = \sum_{k=1}^{7} x_k \zeta_{\mathbf{k}} \eta_{\mathbf{k}}{}^T \tag{10}$$

where E_{ij} is defined as a matrix that has a 1 as the (i,j)th entry, $i, j = 1, 2$, and 0 elsewhere. That is,

$$E_{11} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, E_{12} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, E_{21} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, E_{22} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

We must find the right combination of vectors $\mathbf{u_k}, \mathbf{w_k}, \mathbf{r_k}, \mathbf{s_k}$, and coefficients x_k to find solutions to this problem.

In the method explained in this thesis, we represent this search process as a problem of matching grids, that makes it much easier to conceptualize and solve in an efficient way. We will start from 5×5 grids and combine them into 10×10 grids, then combine those into 20×20 grids,

We begin by identifying all possible matrix products formed by multiplying a pair of v_k vectors. The products $v_k \times v_j^T$ are shown in Figure 1.

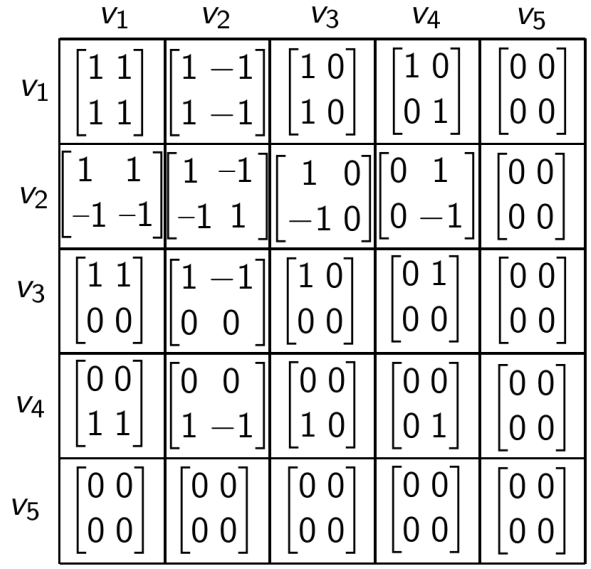

FIGURE 1. A table showing the v_k product matrices.

3.3. **Creation of 5 by 5 grids.** The collection of 5×5 grids is created with a brute force algorithm. All possible combinations are checked, and those that are found to equal an E_{ij} matrix are kept. The idea during this step of the method is to determine all relevant 5×5 grids, i.e., all linear combinations of the 2×2 outer product matrices appearing in Figure 1 that give the matrices appearing in the four blocks on the left hand side of equation (10), for $i, j = 1, 2$. It turns out that this amounts to 111 5×5 grids for each E_{ij}, each one corresponding to a unique linear combination of between 1 and 4 matrices appearing in Figure 1. A 5×5 grid that corresponds to an E_{ij} with more than four terms could be constructed, but we know empirically that no fast matrix multiplication algorithms contain such a grid, and later we argue that it would be impossible mathematically for a 5×5 grid with more than four entries to appear in a fast matrix multiplication algorithm.

Consider the grid shown in Figure 2. The 1 in the (1,2) position corresponds to the outer product $v_2 \times v_1^T = \left[\begin{smallmatrix} 1 & 1 \\ -1 & -1 \end{smallmatrix}\right]$ (see Figure 1), the -1 in the (4,3) position corresponds to the $-v_4 \times v_3^T = -\left[\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right]$ outer product, the 1 in the (3,4) position corresponds to the $v_3 \times v_4^T = \left[\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right]$ outer product, and the 1 in the (4,4) position corresponds to the $v_4 \times v_4^T = \left[\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ outer product. The grid in Figure 2 indicates that

$$(1)v_1 \times v_2^T + (-1)v_3 \times v_4^T + (1)v_4 \times v_3^T + (1)v_4 \times v_4^T$$

$$= \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = E_{11} \tag{11}$$

Thus, the 5×5 grid encodes a solution for the upper left or lower right block of equation (10) for the $i = j = 1$ case.

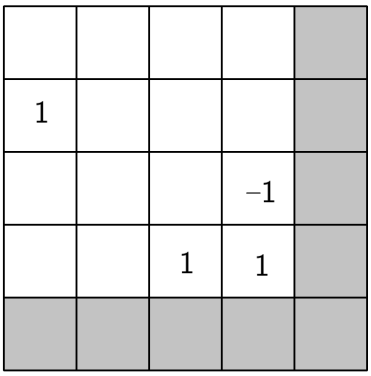

FIGURE 2. A 5×5 grid representation of one E_{11} matrix from Strassen's algorithm. The zero row and zero column are greyed out.

3.4. **Creation of 10 by 10 grids.** Once all of the 5×5 grids have been generated, we can create 10×10 grids. Each 10×10 grid is composed of a pair of E_{ij} 5×5 grids, placed in the upper left and lower right corners, and zero 5×5 grids placed on the off-diagonals. This corresponds to equation (10). See Figure 3 for an example of a 10×10 grid corresponding to Strassen's algorithm. The 5×5 grid in the top left quadrant is the simplest 5×5 grid, as the entry for $v_3 \times v_3^T = E_{11}$, and the 5×5 grid in the bottom left is the one from Figure 2, and this 10×10 grid is one of the grids used in Strassen's algorithm. Since there are 111 different 5×5 grids for each E_{ij} matrix, it would seem that there would be four sets of $111^2 = 12321$ 10×10 grids, one set for each E_{ij}. However, we can eliminate a very large number of these. It turns out that 88 of the 111 5×5 grids for each E_{ij} have four elements. Only one grid contains one element. This means that $88 \times 110 = 9680$ of the 10×10 grids will have a total of six or more elements, leaving 2641 with five or fewer. Empirically, and for other reasons that will be discussed later, we know that if a 10×10 grid contains six or more elements, it cannot correspond to a fast matrix multiplication algorithm, so we can discard these.

For each of the remaining 2641 10×10 grids, we must find all of its different "solution rectangles." The solution rectangles and the rules of determining them are detailed in Chapter 4. It is in determining these solution rectangles that we find solutions to the equation (7) and thus find fast

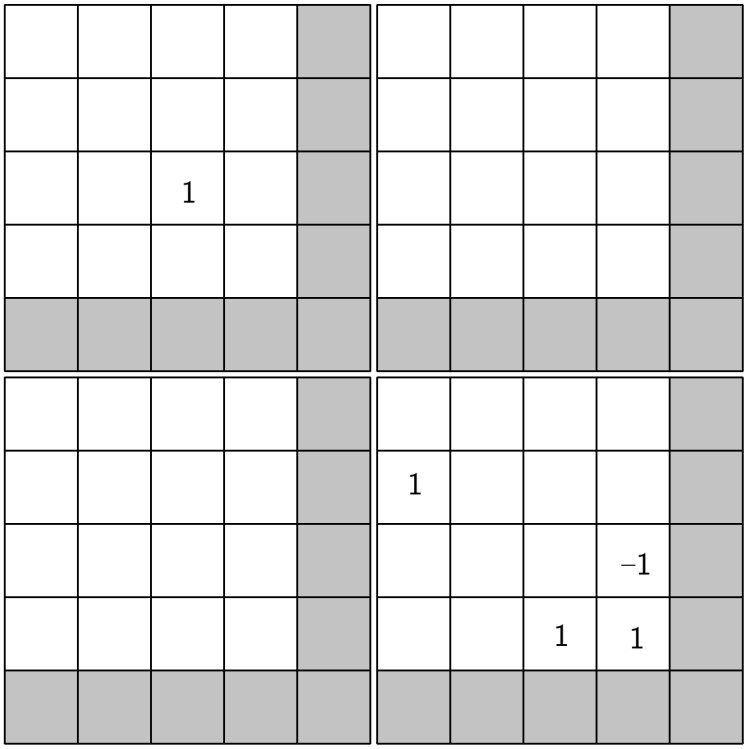

FIGURE 3. A 10×10 grid in Strassen's algorithm. The zero rows and zero columns are greyed out.

matrix multiplication algorithms. The basic idea is that we must identify a set of rectangles having one corner in each quadrant of the 10×10 grid, such that every non-zero grid entry is included in exactly one solution rectangle.

3.5. **Creation of 20 by 20 grids.** To find a 2×2 fast matrix multiplication algorithm, we must construct a 20×20 grid composed of four 10×10 grids which uses only seven different solution rectangles. (We will show later in the thesis how each solution rectangle leads to one M-value; thus the number of distinct solution rectangles determines the number of matrix multiplications performed by the corresponding algorithm.) If a rectangle appears more than once in a 20×20 grid, that is acceptable and the reappearances do not count toward the limit of seven. It only needs to be calculated once, and after that its reuse is "free." For the same reason, the negative of a solution rectangle can be counted as the same rectangle in this regard, since the corresponding M-value can be subtracted instead of added without needing to perform another multiplication.

Once the 10×10 grids have been processed, such that their solution rectangles are in place, they are in four different groups, one for each E_{ij}, $i, j = 1, 2$. Any combination of four of them (one from each group) constitutes a valid 20×20 grid. See Figure 4 for the 20×20 grid that corresponds to Strassen's algorithm.

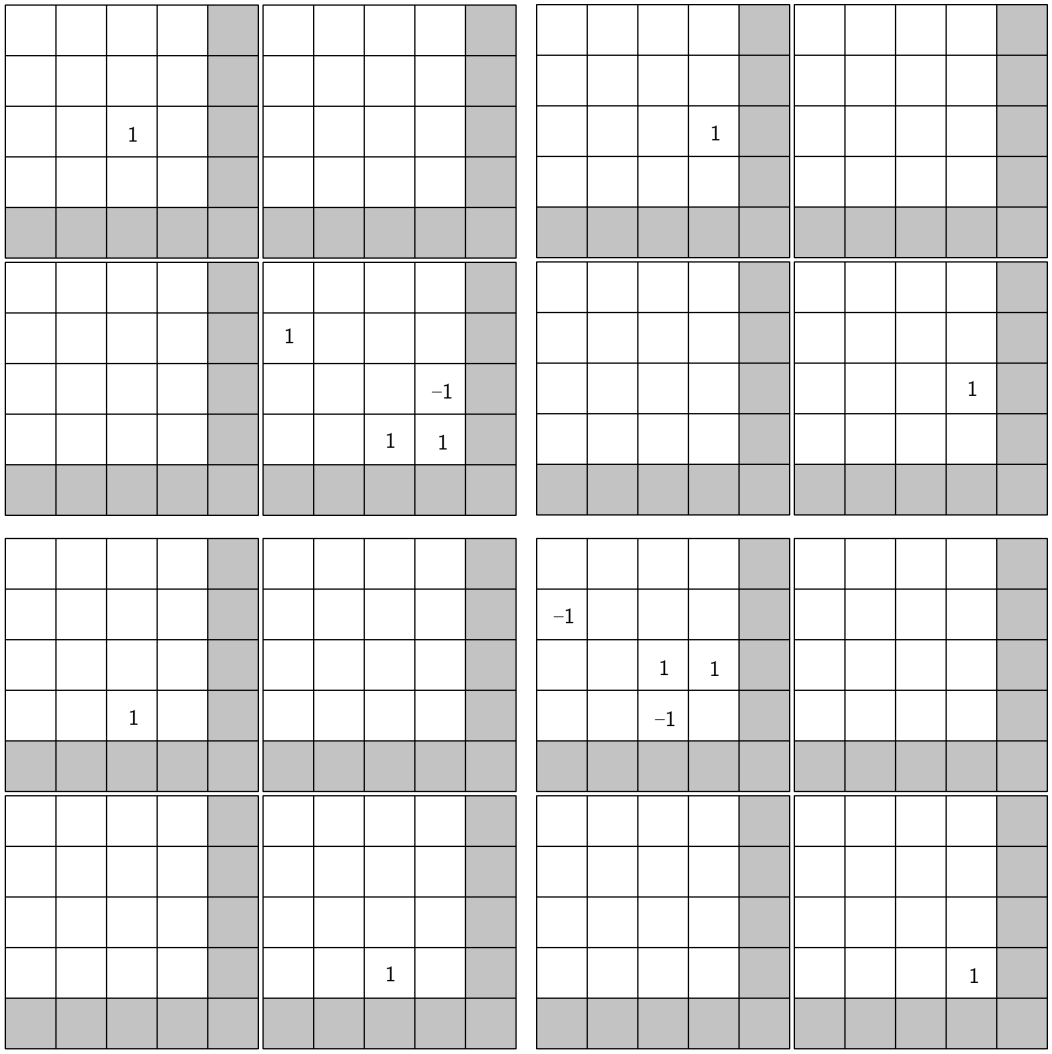

FIGURE 4. The initial 20×20 grid for Strassen's algorithm, without solution rectangles superimposed. The upper left 10×10 grid corresponds to E_{11}; the upper right 10×10 grid corresponds to E_{12}; the lower left 10×10 grid corresponds to E_{21} and the lower right 10×10 grid corresponds to E_{11}.

A 20×20 grid is said to be valid if the algorithm corresponding to the grid would multiply matrices correctly. However, only very specific combinations will give a 20×20 grid that corresponds to Strassen-type algorithms that only use seven matrix multiplications. These combinations are the ones with only seven distinct solution rectangles. Solution rectangles are detailed in Chapter 4, but an example of Figure 4 with solution rectangles in place is given in Figure 5. The twelve solution rectangles correspond to the twelve M-values that are added and subtracted to form the C-values.

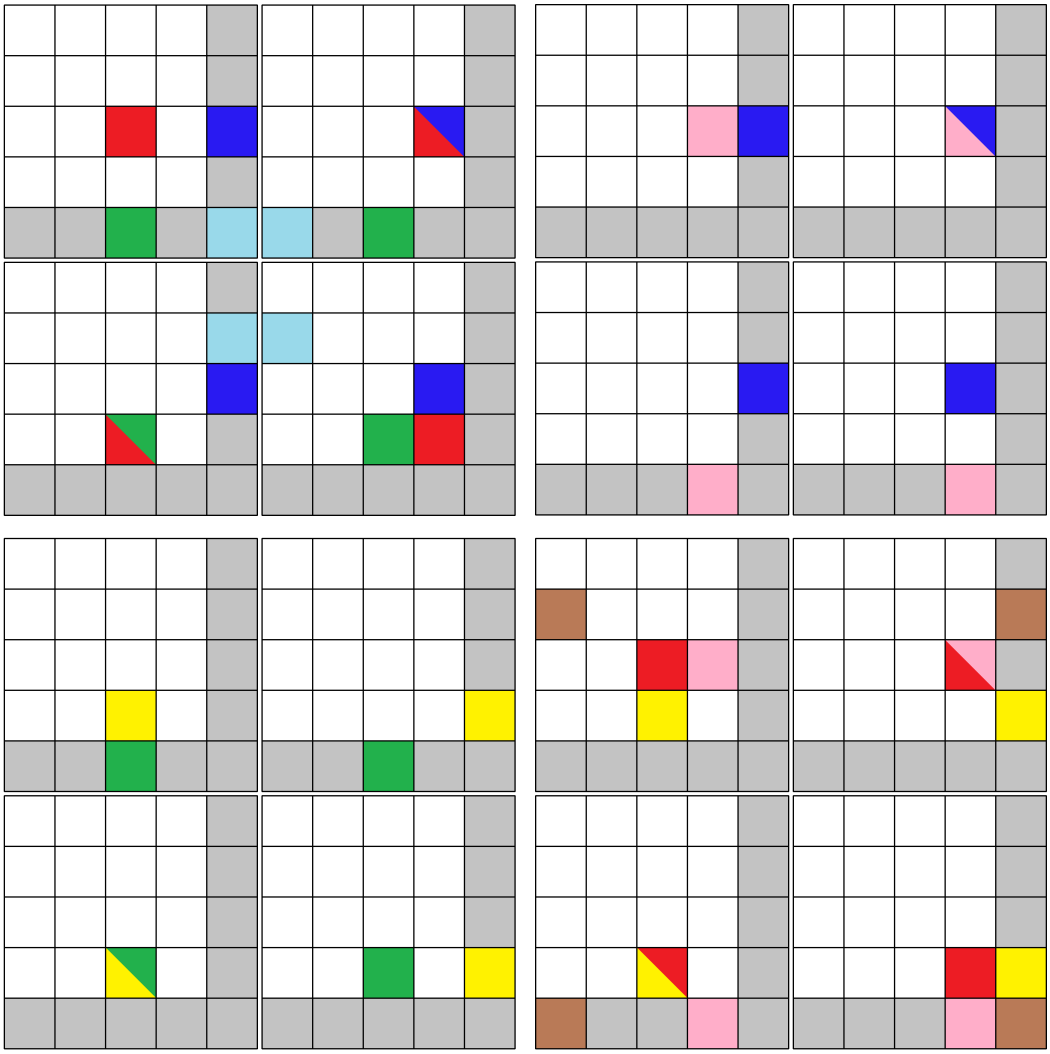

FIGURE 5. The 20×20 grid for Strassen's algorithm, with solution rectangles superimposed. Each solution rectangle is identified by its corners which are all the same colour. Solution rectangles have the same colour when the same rectangle appears in more than one 10×10 grid.

4. Further Extensions and Implementation

We have implemented the algorithm for determining the solution rectangles for a given 20×20 grid (and thus the corresponding M-values, as will be explained later in this chapter) using the Scilab programming environment [10]. The Scilab script consists of a large number of functions, all working together to process the grids. Rather than storing a 10×10 grid as a sparse 10×10 array, we store only the non-zero entries in the array, in the form of a $k \times 2$ array, where k is the number of non-zero entries in the grid. The first entry in each row of this array is the element's index within a 5×5 grid, and the second is its quadrant within the 10×10 grid. The index can be any integer

between -25 and 25, except for 0, and the quadrant will be between 1 and 4 (the quadrants are numbered row-wise from 1 to 4). The indices correspond to a 5×5 grid entry, as in Figure 6, where a negative index indicates a -1 coefficient for the matrix.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

FIGURE 6. The indices of a 5×5 grid.

At first, we tried storing each grid entry with 5 items per row of the array, keeping its index, sign, quadrant, row, and column separate. However, this made it more difficult to create consistency between functions, as they sometimes only needed some of the information. The simple two-item representation makes it easy to keep a consistent data structure. Theoretically, the index and quadrant could have been combined, to form a super-index ranging from -100 to +100 (with 25 indices going through each quadrant sequentially), but this creates more problems than it solves. Quadrants need to be determined frequently, and it is important to see if two elements are in the same place within their respective quadrants.

The most important functions are the matching functions, which are used to find every possible arrangements of solution rectangles. Every other function either prepares data for the matching functions to use, or makes use of the data the matching functions generate.

The main challenge in designing this program was not in finding individual solution rectangles, but in efficiently computing all possible combinations of solution rectangles for a given 10×10 grid. The calculations must be balanced between not duplicating cases (as that is inefficient, and creates more inefficiencies later), but also not missing any possibilities.

By separately identifying the five ways in which elements on a 10×10 grid can be matched together (these five ways are identified later), it becomes possible to ensure that no solution rectangles are missed. Then, instead of creating entire matched grids sequentially, the matching functions were made to return "options." An option is simply one way in which some number of elements could be matched, i.e., arranged into solution rectangles. After running all the matching functions, we will have a three-dimensional array composed of many options, each of which is a subset of the grid containing between one and three solution rectangles.

After all the options have been composed, we face the first time-intensive calculation. Every option that has been calculated will use some subset of the original non-zero grid entries. For a combination of options to be valid, it must include every original non-zero grid entry in a solution rectangle exactly once, and it must have no more than four solution rectangles. A single element can not be part of multiple solution rectangles in a single grid, so we can't choose multiple options that all include the same element. Likewise, all elements must be part of one solution rectangle, so if one element is not included in any options, then that set of options is not complete. This process takes approximately 30 minutes on our hardware, an Intel Core i7 6700k CPU.

This operation is run on every 10×10 grid that can possibly produce useful solutions. At first glance, this would seem to be a very difficult task, but by removing the 10×10 grids with more than five elements as mentioned in Chapter 3, the search space can be massively reduced. The grids removed are the most expensive $\approx 87\%$ of the grids, as a larger number of grid elements corresponds to a much larger amount of time spent on finding its solution rectangles. A 10×10 grid with eight elements takes several minutes to process, but a 10×10 grid with five elements takes at most 15 seconds. By removing these expensive grids, the estimated run time of this function on our hardware is reduced by $\approx 99.93\%$, from approximately one month to thirty minutes.

4.1. **The Rules of Determining Solution Rectangles.** With the 10×10 grids, we must superimpose sets of solution rectangles on to the grids, with one corner in each 5×5 grid (referred to as the quadrants of a 10×10 grid; for our purposes, the top left grid is the first quadrant, top right is the second quadrant, bottom left is the third quadrant and bottom right is the fourth quadrant). These solution rectangles correspond to one of the intermediate M-values of a fast matrix multiplication algorithm. (This will be explained later in the thesis). The process of finding these solution rectangles is called "matching" and a set of solution rectangles created through this process is called "a match." The different types of matches are explained later.

4.2. **Virtual, Zero, and Original Elements.** The entries of a grid, usually called "elements" because of their indivisible nature, can be considered in three categories: original, zero-place, and virtual elements. See Figure 7 for examples of each of these in a 10×10 grid.

An **original element** is one of the non-zero elements that is part of a 5×5 grid. These elements are not allowed to be changed at all in the matching process–they are set in stone.

A **zero element** or "zero-space element" is any element placed in a zero-row or zero-column. These are the freebies of making matches, since every row and column of every quadrant has a zero space available. Every zero-space corresponds to a zero matrix (this can be seen in Figure 1) and so

elements can be safely added to any grid's zero-spaces without changing the value of the grid, i.e., the corresponding linear combination of outer products of v_j matrices; see, e.g. equation (11).

Finally, a **virtual element** is the most complicated type. It is technically two elements: a +1 and -1 placed on the same spot in the grid. Both of these elements are contained separately in two solution rectangles. We need to introduce such grid elements in order to form some of the solution rectangles, because adding and subtracting the same value is the only way to avoid changing the value of the 5×5 grid.

4.3. **The function MatchSolutions.** Despite taking up the other half of the program's run time, this part is far more simple than the part that deals with the solution rectangles. In the naive implementation, we would have four nested loops running through each of the four E_{ij} groups to process the corresponding 10×10 grids. Some of the functions that create the 10×10 grids will create the same grid more than once. After these redundant grids are pruned, there are at most 991 different 10×10 grids in each of these groups (some groups have slightly fewer, but they are all close to 991), so this means that $\approx 991^4 \approx 9.64 \times 10^{11}$ different 20×20 grids must be checked. When our software is run, we find that the true amount is 8.34×10^{11} grids.

Each solution rectangle is indexed as a single integer using a one-to-one function, so a 20×20 grid is represented as a collection of these integers, up to four for each 10×10 grid. We want to find a set of four 10×10 grids with exactly seven different integers (since we know that seven is optimal.)

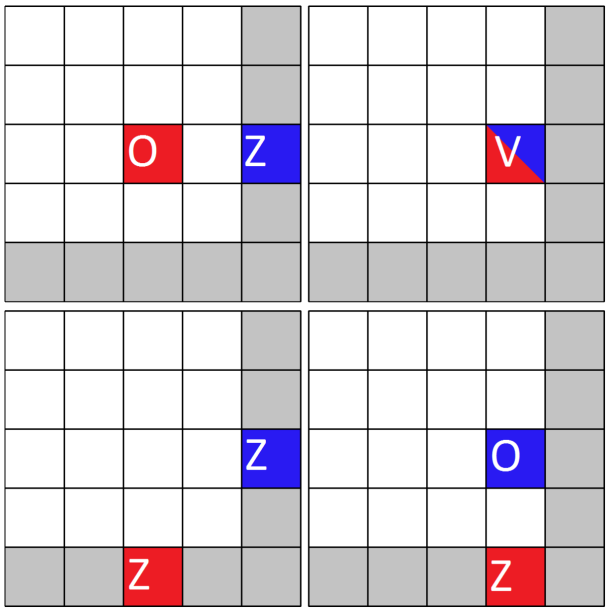

FIGURE 7. A 10×10 grid with two solution rectangles identified by their vertices: one with blue vertices and one with red vertices. Original elements are marked O, zero elements are marked Z, and a virtual element is marked V.

On a high-end home desktop with a Core i7 6700k CPU, it takes approximately 10^{-4} seconds to check a single 20×20 grid, so we would expect that an exhaustive search of these grids would take approximately 8.34×10^7 seconds, or 2.6 years. However, there is a very important optimization that can be done: at every step of the nested loops, we will check to see if the current solution has more than seven different solution rectangles already.

The majority of 10×10 grids have four original elements in one quadrant, which means that the majority of 10×10 grids must have four solution rectangles. Since the majority of 10×10 solutions have four solution rectangles, and equivalent solution rectangles are relatively infrequent (there are thousands of different solution rectangles that are distributed in a relatively uniform way), it is likely that each additional 10×10 grid in a 20×20 grid will add four new solution rectangles. Because of this, the majority of 20×20 grid constructions will abort after just two 10×10 have been considered (since 8 distinct solution rectangles would correspond to a multiplication algorithm with 8 multiplications, which implies that the algorithm is of no interest.) Whenever this occurs, every set of 10×10 grids that starts with the two 10×10 grids as described above is skipped, which is a savings of $991^2 = 982081$ 20×20 grids.

When this optimization is implemented, $\approx 99.996\%$ of the 8.34×10^{11} grids are pruned early. Only 3.64×10^7 are completed down to the "bottom" inner loop, and this takes slightly over an hour to complete.

4.4. **Dealing with the Negatives.** Every element in a grid can be a zero, one, or negative one. A large amount of complexity is added by the possibility of negatives. One important fact is that every solution rectangle must contain an even number of negatives. This makes sense, as we do not have direct control over the indices. By choosing the vectors $\mathbf{u}$, $\mathbf{w}$, $\mathbf{r}$, and $\mathbf{s}$ (from eq. (10)), we are choosing two columns and two rows in the 10×10 grid, and we mark the matrix each pair represents as an element in the grid. When we make any one of these vectors negative, it will switch the signs of two elements in the solution rectangle. This makes it impossible for a solution rectangle to have one or three negatives.

In the function MatchSolutions, the amount of computing expended on the construction of 20×20 grids depends on the set of solution rectangles that are being checked for identical entries. The only way to reduce these computational costs is to reduce the size of the set. One way to accomplish this is to remove negatives of zero-space elements and virtual elements entirely. But why are we allowed to do this? There are two reasons. For the zero-space elements, the negative is already an abstraction. The matrix $\left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ is equivalent to the matrix $\left(\begin{smallmatrix} -0 & -0 \\ -0 & -0 \end{smallmatrix}\right)$. This means that if we disregard the negative, it won't change the algorithm at the end; the only thing we need to be concerned about is making the solution rectangle invalid by having an odd number of negatives.

We can reintroduce the negatives at the end by assigning the negative and positive element of each virtual element to a solution rectangle in some valid way, and then assuring that each solution rectangle is valid by changing the signs of the zero elements (which can be changed independently). When the solution rectangle has no zero elements it is more complicated but still always possible to arrange the signs in some valid way. This will give one representative member of the set of all algorithms that could be made by some variation of a 20×20 grid's negative entries.

For virtual elements, it is a bit more complicated. There are two varieties of solution rectangles that will include a virtual element. The first one is one original element, one virtual element, and two zero-elements, which can be seen in Figures 8, 9, and 10 respectively. The second is two virtual elements and two original elements, which can be seen in the red solution rectangles of Figures 11 and 12. Every virtual element is a 1 in one solution rectangle and a -1 in the other, so we do not have arbitrary control over how we choose them. However, if we ignore negatives until the very end, we will never accidentally create an arrangement where we can not reconcile the negatives. Every solution rectangle of the second type has both virtual elements "cleaned up" by solution rectangles of the first type (e.g., Figure 8). Since we do have arbitrary control over the choice of the sign of the first type, then any of the second type can be made valid.

4.5. **On negative priorities:** There are many scenarios where the program would have the option of how to arrange the negatives. When an element is in a zero-row or zero-column, its entries are $\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}$ no matter which zero-space index it has. Whether or not it is negative is essentially an abstraction. Because of this, we ignore its sign and count it as positive in the construction of the 20×20 grids. At the end of the program, when the algorithms are being printed in a human-readable format, the program simply chooses one way of representing that particular algorithm.

When matches are being made, negatives are kept despite it not being strictly necessary. This is partially because before software was finished it was not clear that negatives could be ignored in this way, but in troubleshooting it acts a directive to show where problems are occuring, because incorrect negatives are relatively easy to spot. Though there is usually a large variety of negative arrangements that could be made with a given solution rectangle, only one will be chosen. Sometimes only some arrangements are possible, so the arrangement that is used depends on a hierarchy. The actual order of the hierarchy is entirely arbitrary, but we put them in this order: same, diagonal, row, column (where "same" means all elements are the same signs, "diagonal" means that elements on diagonals share a sign, and so on). If multiple options are available, we simply use the one that appears first in the hierarchy.

4.6. **Types of Matches.** There is a limited number of ways in which the elements of a 10 by 10 grid may be matched, i.e., identified with a set of solution rectangles. By checking for each match in turn, all possible matches are generated. At first we attempted to design a general function to identify any match, but this quickly becomes a daunting task as one must deal with varying numbers of elements, varying negatives, and varying combinations of arrangements.

For this section, the signs of the grid elements do not matter, so we will represent matches with coloured grid entries. All entries of the same colour form the vertices of one solution rectangle. Where one entry has two colours, that is a virtual element.

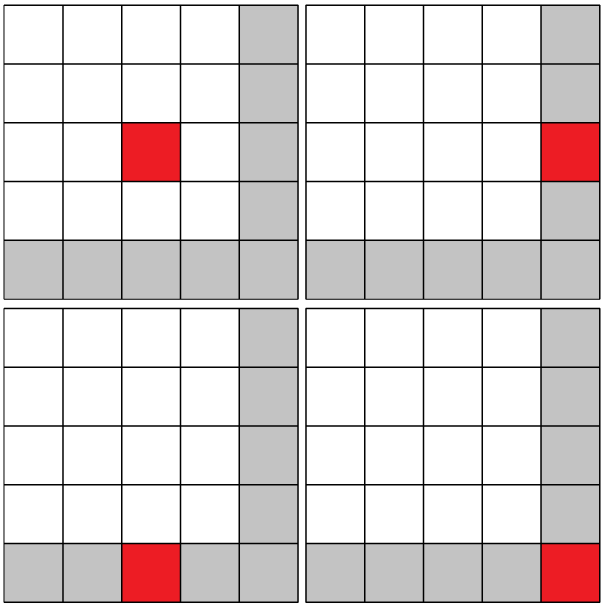

FIGURE 8. An example of a type 0 match. For readability, the zero rows and zero columns are given a grey background.

Type 0 matches, or "Solo matches": A solo-match is the simplest kind of match, and as the name may indicate, the only one that requires one element. It does not match the non-zero element with any other original non-zero elements. A solo match is simply the original element, included in a solution rectangle with one entry from the zero column in the horizontal quadrant, one entry in the zero row of the vertical quadrant, and one entry in the zero-corner of the opposite quadrant. These are usually only used to "clean up" whatever original elements are left over after making more complex matches. Figure 8 gives an example of a type 0 match.

Type 1 matches: The type 1 match is also fairly simple, and can be made with any pair of elements from diagonally opposite quadrants. Each type 1 match comes in four varieties. Refer to Figure 9 to see a type 1 match with the intersection of the red and blue rectangles appearing in the top right quadrant. A type 1 match between two elements puts a virtual element at their intersection

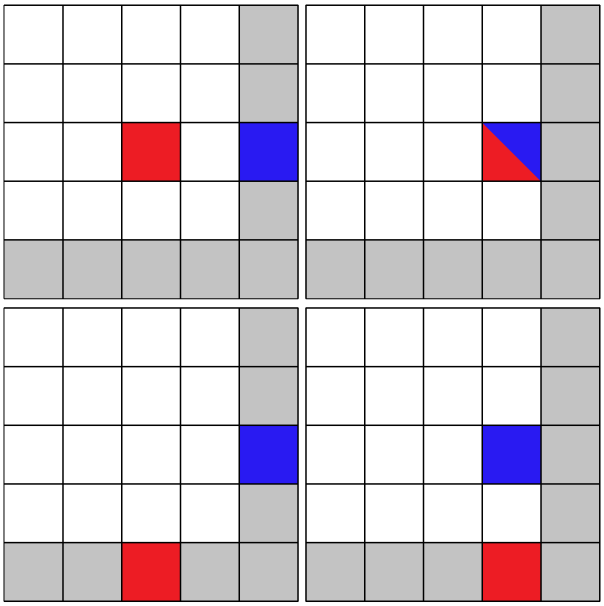

FIGURE 9. An example of a type 1 match.

in one of the empty quadrants: the top right or bottom left. Then, each original element is put into one of the two solution rectangles associated with this virtual element. Depending on which original element is placed in a solution rectangle with the positive part and which is placed with the negative part, we will have different solution rectangles. Then, the remaining elements of the solution rectangles are placed in the appropriate zero-rows or zero-columns, and negative signs are assigned as appropriate to the hierarchy of negative arrangements explained in the previous chapter. The remaining two variations arise from putting the virtual element on the other intersection of the two original elements, and varying the negatives again.

Type 2 matches: See Figure 10 for an example of a type 2 match. These begin to become more complicated, and have more stringent requirements. In order to make a type 2 match, one quadrant must have two elements in the same row or the same column. Then, a virtual element is placed in one of the four non-zero locations in the other 5 by 5 grid along this row or column, and the two original elements match up with it, and the remaining places are filled by zero-entries in the remaining two columns. By varying arrangements of negatives, or by varying which of the four locations along the row/column the virtual element is placed in, each type 2 match contains eight variations.

Type 3 matches: The type 3 match is the first match that has more original elements than solution rectangles, and this is the key for fast matrix multiplication algorithms, including Strassen's algorithm. The type 3 match has three solution rectangles and four non-zero original elements. Figure 11 gives an example of a type 3 match. In the first quadrant, there must be three elements,

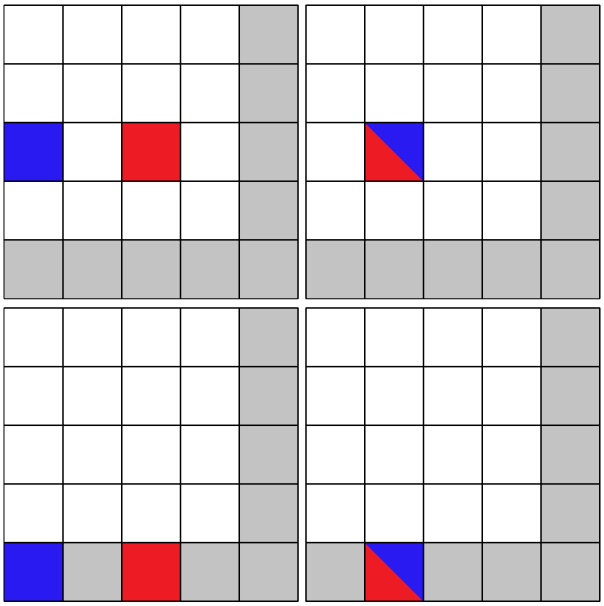

FIGURE 10. An example of a type 2 match.

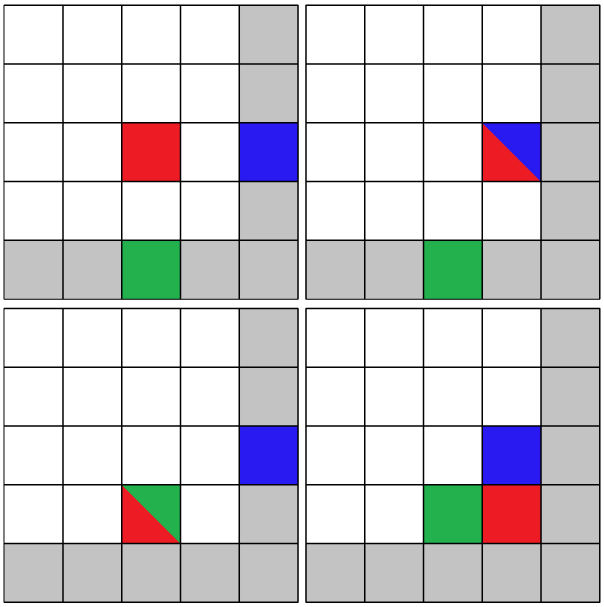

FIGURE 11. An example of a type 3 match. Note that the red solution rectangle is an efficient solution, as it does not include zero elements.

such that one of these elements shares a row with another, and a column with the third. There is no restriction on the element in the opposite quadrant. Then, by creating two virtual elements and "cleaning them up" with the two other elements, the intersecting element from the first quadrant is matched with any element in the opposite quadrant. Note that the red solution rectangle in Figure 11 contains two original elements, which has not been done in any previous match.

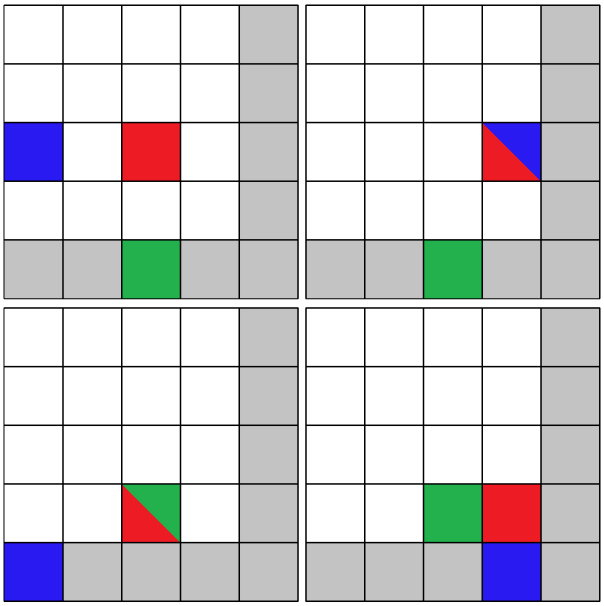

FIGURE 12. An example of a type 4 match.

Type 4 matches: This type of match is similar to the type 3 match, in that it involves one efficient solution rectangle with no zero-elements. However, instead of having one quadrant with three elements and another with one, this type has two elements in each quadrant, and either both pairs of elements are row-aligned or both are column-aligned. Since no element is "central," like the intersection in type 3, this has a larger number of varieties in the arrangement of virtual elements. Figure 12 gives an example of a type 4 match.

We can empirically verify that these are the only kinds of matches, as they are the only ones that appear in any 2×2 fast matrix multiplication algorithms. But also we can argue theoretically that it would not make sense to have any others.

All matches (other than type 0) are based on the idea of creating virtual elements (a +1 and -1 on the same grid entry) to make one solution rectangle, and then "cleaning them up" with a second solution rectangle. A virtual element can be "cleaned up" by being included in a solution rectangle that includes an original grid element that it shares a column or row with it. If there is one virtual element, then there must be two original elements making use of it in two matches (this is what we see in the type 1 and type 2 matches). If there are two virtual elements, then it is possible to create a match using two original elements and two virtual elements (the red matches in Figures 6 and 7), and then "clean them up" with two more matches. This is what gives us type 3 and type 4 matches. It is impossible to come up with a new kind of match that is not just a combination of these smaller matches.

Why is this? Consider our limitations: we can have at most five original non-zero elements (as argued earlier in this thesis), with at most four in one 5×5 grid. A match (other than a type 0) can only include an even number of original elements. For every virtual element it is either matched with two original elements that are then matched with zero rows/zero columns (as in type 1 and type 2 matches) or it is matched with two original elements which are each matched with a second virtual element, and these two virtual elements must be matched with a third and fourth original element. To have more than two virtual elements, we would need six original elements, which we know is not possible. Any match made without virtual elements is just a series of unrelated type 0 matches. This means that any new matches would have to have one or two virtual elements. It is easy to see that any match with only one virtual element is either a type 1 match or a type 2 match. A match with two virtual elements that would not be a type 3 or type 4 match would require that all four of the original elements in the match to be in the same quadrant. This is a possibility, and such a match could be made, but it would actually just be two type 2 matches.

With the same empirical knowledge of other 2×2 fast matrix multiplication algorithms, we know that no non-trivial solutions for the top right and bottom left 5×5 grids of a 10×10 grid are used, only trivial solutions (i.e., the matrices that correspond to zero elements and virtual elements), so our five types of matching algorithms can all assume that the top right and bottom left grids do not contain any "free" elements to be used in solution rectangles.

4.7. **From grids to algorithms:** Once the software has run and the 20×20 grids are completed, i.e., a set of seven distinct solution rectangles are superimposed on the grid, see, e.g., Figure 5, we must convert the 20×20 grid back into the format of M-values and C-values. We will perform this operation on Strassen's algorithm, using the grid shown in Figure 5. The signs of these elements are not shown, but for our example we will use a solution rectangle with four positive elements.

Each solution rectangle of a distinct colour corresponds to one of the M-equations of the Strassen algorithm. The process of translating a solution rectangle into an M-equation is straightforward.

Recall the four vectors from eq. (10): $\mathbf{u_k}, \mathbf{w_k}, \mathbf{r_k}, \mathbf{s_k}$. Each is a set of vectors from v, eq. (9), corresponding to one row or column used by a solution rectangle: $\mathbf{u_k}$ corresponds to the top row, $\mathbf{w_k}$ to the bottom row, $\mathbf{r_k}$ for the left column, and $\mathbf{s_k}$ for the right column.

Once we determine these vectors, we then create the matrix P_k, such that

$$P_k = \left[\begin{array}{c} \mathbf{u_k} \\ \hline \mathbf{w_k} \end{array}\right] \left[\begin{array}{c|c} \mathbf{r_k} & \mathbf{s_k} \end{array}\right].$$

Let us first consider the red solution rectangle, which appears in the top left and bottom right quadrant. Its top elements are on the third row, so $\mathbf{u_1} = v_3$. Its bottom row is the fourth row, so

$\mathbf{w_1} = v_4$; its left column is the third column, so $\mathbf{r_1} = v_3$; its right column is the fourth column, so $\mathbf{s_1} = v_4$. Thus,

$$P_1 = \left[\begin{array}{c} \mathbf{u_1} \\ \hline \mathbf{w_1} \end{array}\right] \left[\begin{array}{c|c} \mathbf{r_1} & \mathbf{s_1} \end{array}\right],$$

$$P_1 = \left[\begin{array}{c} v_3 \\ \hline v_4 \end{array}\right] \left[\begin{array}{c|c} v_3 & v_4 \end{array}\right],$$

$$P_1 = \left[\begin{array}{c} 1 \\ 0 \\ \hline 0 \\ 1 \end{array}\right] \left[\begin{array}{cc|cc} 1 & 0 & 0 & 1 \end{array}\right].$$

.

When we reintroduce **a** and **b**, we see that this is the same representation of M-values we saw in eq. (4): e.g.,

$$m_1 = \mathbf{a}^T \left[\begin{array}{c} 1 \\ 0 \\ \hline 0 \\ 1 \end{array}\right] \left[\begin{array}{cc|cc} 1 & 0 & 0 & 1 \end{array}\right] \mathbf{b}.$$

And by multiplying out this equation we recreate the standard representation of m_1 from Strassen's algorithm, $m_1 = (a_{11} + a_{22}) \times (b_{11} + b_{22})$. The equation for each M-value is determined by following this same series of steps for each solution rectangle in the 20×20 grid.

The calculation of the C-values is determined by which solution rectangles appear in each 10×10 quadrant of the 20×20 grid. In this case, the other solution rectangles in the top left quadrant turn out to give m_4, $-m_5$, and m_7, thus $c_{11} = m_1 + m_4 - m_5 + m_7$.

When a solution rectangle and its negative appear in a 20×20 grid, determining which solution rectangle is m_k and which is $-m_k$ is arbitrary. There is no practical reason for one to be chosen over the other.

4.8. **Functions:** This section gives a brief description of the Scilab functions we have developed.

4.8.1. *One through FourTermSolutions and allTermSolutions.* A 5x5 grid may have between 1 and 4 elements (or terms). For reasons explained earlier in the thesis, no 10×10 grid may have more than 5 elements, and a 10×10 grid contains two 5×5 grids, each of which must contain at least one element, so a 5×5 grid can contain at most four elements. These functions find every 5×5 grid, with each function finding grids with a different number of terms.

OneTermSolutions just returns a hard-coded list, as there are only four of these solutions. The other functions iterate through all possibilities and discard the invalid 5×5 grids. AllTermSolutions simply runs all the functions OneTermSolutions through FourTermSolutions functions sequentially and combines the output to form a single list.

4.8.2. *SetNeg.* This is a helper function that is used in most of the FindTypeXMatch functions. It is a very simple function that changes the sign of the first entry in a vector with two components. This is so that the software can easily change the sign of an element index in the same line that it assigns it to a variable, which makes the matching functions more readable (since many have signs which depend on other entries or calculations in the match).

4.8.3. *FindSoloMatch.* The calculation for a type 0 match is very simple. It takes a single grid element as an argument, and returns an options matrix with the vertices of the solution rectangle made from taking the zero-row and zero-column entries that share a column or share a row with the element passed to the function. If the original element is negative, then by our negative priority every element of the match will be considered negative.

4.8.4. *FindTypeOneMatch.* Between any pair of elements, there are four type one matches that can be made: one with the virtual element in quadrant 2, and one with the virtual element in quadrant 3, each one having a second variety with a different arrangement of negatives. Given a list of elements, this function will take the one at the top of the list and create type 2 matches with every element in the opposite quadrant. To find every type 1 match, the main function will call FindTypeOneMatch on a shortening list (elements 1 through 5, elements 2 through 5, *etc.*).

4.8.5. *FindTypeTwoMatch.* A type 2 match is more selective than a type 1 match. A type 2 match requires two elements in the same quadrant in the same row or in the same column. There are eight separate matches made for every such pair of elements, one for each potential location of the virtual element in the quadrant along the row or column, each one having two possible arrangements of negatives.

4.8.6. *FindTypeThreeMatch.* The type 3 match is more complicated than the earlier types, due to the asymmetry of the elements involved. The element that is at the intersection of two others in its quadrant plays a different role in the match than other elements, so we first run a helper function "t3dmReorder" to rearrange the elements in an order based on their role in the match, so that the function can access each one, knowing its position. A key difference between this function and the other matches is that the calculations of the solution rectangles are made more generic and configured based on an analysis of the elements that have been passed to the function. Conversely, in

the functions for type 1 and type 2 matches, there are different blocks of code for different variations in the match.

4.8.7. *FindTypeFourMatch.* The type four match is similar to the type three match, but there are a larger number of variations that make the generic code more complex. Firstly, the elements may be arranged as sharing columns or sharing rows, and the algorithmic details for each of these cases are different enough that it made sense to break the function into two blocks of code. It also involves some of the most complex calculations for the individual elements. The most egregious example of this complexity appears in the case of the elements sharing columns, where the calculation of the 9th element in the options matrix involves taking the ceiling of the sum of the floor of the modulo of one element and another element divided by five. Though it took some time to troubleshoot, this complexity pays off in the long run due to being applicable to many variations of the match. Having eight separate blocks of code based on the specifics of the match was a choice we could have made, but this was deemed more difficult to maintain.

4.8.8. *allTenByTenCombinations.* This function receives as input every possible original 5x5 grid and combines them in every way that could possible give a productive result.

No 10×10 grid may have more than 5 elements, because the most efficient types of matches can create (at best) 3 solution rectangles from 4 elements. Therefore, if there are 6 or more elements there will be 5 or more solution rectangles, and it is known that no fast matrix multiplication algorithm has more than 5 solution rectangles in one quadrant. Since grids with more elements take more time to process, and more of them exist, this pruning step is very important, reducing the runtime from months to an hour.

4.8.9. *ProcessSolutions.* When the collection of 10×10 grids has been created, they are in a sparse format in a four-dimensional array. Most 10×10 grids have less than a dozen variations, but a very small number have as many as 169 variations, so the entire array must have a size of 169 in one of its dimensions. Since the process of matching these 10×10 grids into 20×20 grids does not need to know which 10×10 grids are variations of others, and every empty array entry slows the function by some amount, the output is processed into a more efficient format. This function rearranges the 10×10 grids into a three-dimensional array with far fewer empty entries.

4.8.10. *Indexify, deindexify, reindexify, indexify2, deindexify2.* A solution rectangle can be defined by the index of its top left vertex and the index of its bottom right vertex, with an additional factor to differentiate between negatives. A solution rectangle's index is calculated based on these three factors, and an automated test confirms that with the function deindexify (which reverses this process), every possible solution rectangle can be stored and retrieved in this way with no collisions.

That is, for any solution rectangle R, R = deindexify(indexify(R)), and for any integer N, N = indexify(deindexify(N)).

Indexify2 and Deindexify2 are similar in function, except that they ignore negatives in zero-spaces and in virtual elements and encode to a smaller set of integers. Before the construction of the 20×20 grids, where non-essential negatives create redundant matrices (as explained earlier in this thesis), solution rectangles are indexed by Indexify2. Negatives are reintroduced in the final stage.

The function reindexify translates any index encoded by indexify to the equivalent solution rectangle as it would be encoded by indexify2. There is no "unindexify" function, as the mapping from indexify to indexify2 is onto, and multiple "indexify" indices can be represented by the same "indexify2" index. This collision will occur with two functions when their arrangement of elements is the same, but one has different signs on the zero elements than the other.

5. Summary, conclusions, and Future Work

5.1. **Summary and Conclusions.** The discovery of fast matrix multiplication algorithms for the 2×2 case is complete. The recent paper by Oh and Moon [2] describes a genetic search in which all four families of Strassen-type algorithms for fast matrix multiplication are identified. However, studying the 2×2 algorithms and studying how to find them in an efficient way is an excellent testing ground for methods of finding improved fast matrix multiplication algorithms for 3×3 matrices. It is conjectured that there are potential improvements in 3×3 matrix multiplication algorithms (the current known smallest number of multiplications for the 3×3 case is 23), but the search space is overwhelmingly large, so a very efficient method for searching for new 3×3 fast matrix multiplication algorithms will be necessary.

Our method abstracts a linear algebra representation of the Strassen-type algorithm into a set of sparsely populated grids, in three tiers: 5×5, 10×10, and 20×20, where each is made of four of the smaller grid. By superimposing rectangles within 10×10 grids with a corner in each 5×5 grid, and aiming to have similar rectangles appear between 10×10 grids, a fast matrix multiplication algorithm is defined by a single 20×20 grid with superimposed rectangles. By restricting the creation of these superimposed rectangles with certain rules, we limit the number of possible 20×20 grids, which reduces the search space, and allows the final brute force search to be completed in about an hour.

The pruning process considered in this thesis removes an enormous number of potential cases. There are so few remaining that they can be processed by a Scilab script in about an hour on our hardware (an Intel Core i7 6700k CPU) to find the remaining valid algorithms. The representation

of the Strassen-type algorithms considered in this thesis may also make it easier to conceptualize the method, so that other pruning or optimizations can be more easily found for the 3×3 case.

With this method as a starting point, it may be possible to search the entirety of the 3×3 case. If not, a heuristic approach may be enough to find just one algorithm (that uses fewer than 23 multiplications). This method can be designed to run in parallel very easily, so brute force calculations may be an effective option.

5.2. **Future Work.** The entirety of our program for the 2×2 case runs in approximately ninety minutes on a computer (Intel Core i7-6700k CPU). This time is split as 30 minutes spent constructing each of the 10×10 grids and 60 minutes in checking all possible 20×20 grids to determine if they represent one of the algorithms we are searching for. In the 3×3 case, the construction of the 42×42 grids (equivalent to the 10×10 grids) grows at least cubically as the number of smaller grids (14×14 for the 3×3 case) increases, since the 42×42 grids are created by choosing every set of three 14×14 grids (it may be possible to apply similar pruning techniques as we used on 5×5 grids, but the cubic growth rate is a maximum, so we will use it for the estimations.)

The construction of 126×126 grids (equivalent to the 20×20 grids) will have a much greater increase in cost. In the 2×2 case, the number of 20×20 grids grows at an approximately quadratic rate as the number of 10×10 grids increases, due to the optimization that aborts nearly all 20×20 grid constructions after two 10×10 grids are included. Conversely, in the 3×3 case, each 42×42 grid adds at most seven solution "rectangles."

If we are searching for a 22-multiplication algorithm, then it will take at least four 42×42 grids being added before the maximum number of solution rectangles is exceeded, which would mean the growth rate would be approximately quartic. If we're searching for an algorithm with 19, 20, or 21 multiplications then it will take at least three 42×42 grids, which gives a cubic growth rate. If it is the case that no algorithms with less than 22 multiplications exist, this calculation will be prohibitively expensive. If this is not the case, and a 19, 20, or 21 multiplication algorithm exists, then the cost will still be a great increase from the 2×2 case, but not to the same degree.

In short, the number of 42×42 grids is equal to at most the cube of the number of 14×14 grids, and the number of 126×126 grids that will be analyzed by the software is approximately the cube of the number of 42×42 grids (for a 19, 20, or 21 multiplication algorithm). So we can approximate the number of checks that will need to be done to be n^6 for n 14×14 grids.

Based on this growth rate, we can roughly estimate the cost of searching for 3×3 fast matrix multiplication algorithms. First, we if we assume the number of 14×14 grids is equal to the number of 5×5 grids, we can calculate that $111^6 \approx 1.8 \times 10^{12}$ 126×126 grid comparisons will be made in the final step of the algorithm. In Scilab, each comparison took 10^{-4} seconds, so this would take slight

less than 3 years. This is only a rough estimation, and the true figure will likely be higher as we are optimistically assuming that there are not significantly more 14×14 than there were 5×5 grids. Considering Scilab's slowness as a scripting language and the effectiveness of parallel computation with the method described in the thesis, such a computation would not be out of reach for a super computer or distributed computing system unless the number of 14×14 grids is many times larger than the number of 5×5 grids.

Even if the calculation turns out to be prohibitively expensive, there are more options for optimization. For one, we would only need to find one algorithm with less than 23 multiplications, of which there may be many, whereas in this thesis we were aiming to find every algorithm to show the effectiveness of the search method. There are many indications that certain arrangements are more favorable than others, and these could be checked preferentially, potentially saving us from needing to investigate the entire search space.

Additionally, we know empirically that all fast multiplication algorithms make use of at least one "minimal" grid (such that each of the subgrids contains only a single element), on which there are only a handful of variations. This means we can run one loop a dozen or so times, rather than over the full number of grids available. Without knowing which grid will contain the minimal grid, we must split the loop into nine separate parts, but there is still an improvement.

The creation of 42×42 grids in the 3×3 case will use the same five types of matches described in this thesis, but it will be between three different pairs of 14×14 grids, as there will be three 14×14 grids that make up the 42×42 grid, from which three different pairs can be taken. Since the 14×14 grids seem to still be restricted to four entries per 14×14 grid (judging by the currently known 3×3 algorithms), the same restrictions on the types of matches will remain. Because of this, writing software to perform the search for a fast 3×3 algorithm should be straightforward.

References

[1] V. Strassen, *Gaussian Elimination is not optimal*, Numer. Math., **13(4)**, 354-356, Sept., 1969.

[2] S. Oh and B. Moon *Automatic Reproduction of a Genius Algorithm, Strassen's Algorithm Revisited by Genetic Search*, IEEE Transactions on Evolutionary Computation, **14(2)**, 246-251, April, 2010.

[3] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic programming*, J. Symbolic Comput., **9(3)**, 251-280, March, 1990.

[4] J. Laderman, *A noncommutative algorithm for multiplying 3 × 3 matrices using 23 multiplications*, Bull. Amer. Math. Soc. **82(1)**, 126-128, Jan. 1976.

[5] F. Le Gall, *Powers of tensors and fast matrix multiplication*, Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation, Jan. 2014.

[6] M. Paprzycki and C. Cyphers, *Using Strassen's matrix multiplication in high performance solution of linear systems*, Computers & Mathematics with Applications, **31(4-5)**, 55-61, February, 1996.

[7] B. MacAdam and G. MacDonald, *An Exhaustive Search for Strassen-type Matrix Multiplication Algorithms*, preprint, 2016.

[8] M. Bläser, *On the complexity of the multiplication of matrices of small formats.* J. Complexity, **19(1)**, 43-60, 2003.

[9] S. Winograd, *On multiplication of 2 × 2 matrices*, Linear Algebra and its Applications, **4(4)**, 381-388, October, 1971.

[10] Scilab Enterprises (2012). *Scilab: Free and Open Source software for numerical computation* (64-bit Windows, Version 5.5.2) [Software]. Available from: http://www.scilab.org

6. Appendix: Source Code

```
function[EList] = oneTermSolutions()
    EList = [11,  0, 0, 0, 0, 1;
             12,  0, 0, 0, 0, 2;
             15,  0, 0, 0, 0, 3;
             16,  0, 0, 0, 0, 4];
endfunction

function[EList] = twoTermSolutions()
    vects = [ 1  1  1  0; //a much smaller set of vectors
              1 -1  0  1];
    m = 0; //counts up the matrices
    hmat = hypermat([2, 2, 16]); //a much smaller set of matrices
    for i = 1:4
        for j = 1:4
            m = m+1;
            tempMatrix = vects(2*i-1:2*i)*vects(2*j-1:2*j)';
            hmat(1:2,1:2,m) = tempMatrix;
        end;
    end;
    E11 = [1,0;0,0];
    E12 = [0,1;0,0];
    E21 = [0,0;1,0];
    E22 = [0,0;0,1];
    x = 1; y = 0;
    A = 1; B = 2; C = 3;

    while A < 15
        for i = 0:1 //two combinations of +/-
            //add and subtract matrices in various combinations
            tempMatrix = hmat(1:2,1:2,A) + ((-1)^(i))*hmat(1:2,1:2,B);
            //Eij it is and if it's negative
            //1 = E11, 2 = E12, 3 = E21, 4 = E22
            //February 2016 update: EList(x,5) is superfluous now but it's easier to just leave it in
            if tempMatrix == E11 then
                EList(x,1) = A; EList(x,2) = ((-1)^(i))*B; EList(x,5) = i;
                EList(x,6) = 1; x = x+1;
            end;
            if tempMatrix == E12 then
                EList(x,1) = A; EList(x,2) = ((-1)^(i))*B; EList(x,5) = i;
                EList(x,6) = 2; x = x+1;
            end;
            if tempMatrix == E21 then
                EList(x,1) = A; EList(x,2) = ((-1)^(i))*B; EList(x,5) = i;
                EList(x,6) = 3; x = x+1;
            end;
            if tempMatrix == E22 then
                EList(x,1) = A; EList(x,2) = ((-1)^(i))*B; EList(x,5) = i;
                EList(x,6) = 4; x = x+1;
            end;
            if tempMatrix == (-1)*E11 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(i+1))*B; EList(x,5) = i+2;
                EList(x,6) = 1; x = x+1;
            end;
            if tempMatrix == (-1)*E12 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(i+1))*B; EList(x,5) = i+2;
                EList(x,6) = 2; x = x+1;
            end;
            if tempMatrix == (-1)*E21 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(i+1))*B; EList(x,5) = i+2;
                EList(x,6) = 3; x = x+1;
            end;
            if tempMatrix == (-1)*E22 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(i+1))*B; EList(x,5) = i+2;
                EList(x,6) = 4; x = x+1;
            end;
        end; //for loop

        //iterate to the next unique set of matrices:
        if B < 16 then
            B = B + 1; //advance B and reset others
        elseif A < 15 then
            A = A + 1; //advance A and reset others
            B = A + 1;
        end;
    end;
endfunction
//16 results (4*4)
function[EList] = threeTermSolutions()
    vects = [ 1  1  1  0; //a much smaller set of vectors
              1 -1  0  1];
    m = 0; //counts up the matrices
    hmat = hypermat([2, 2, 16]); //a much smaller set of matrices
    for i = 1:4
        for j = 1:4
            m = m+1;
            tempMatrix = vects(2*i-1:2*i)*vects(2*j-1:2*j)';
            hmat(1:2,1:2,m) = tempMatrix;
        end;
    end;
    E11 = [1,0;0,0];
```

```
    E12 = [0,1;0,0];
    E21 = [0,0;1,0];
    E22 = [0,0;0,1];
    x = 1; y = 0;
    A = 1; B = 2; C = 3;

    while A < 14
        for i = 0:3 //four combinations of +/-
            //add and subtract matrices in various combinations
            tempMatrix = hmat(1:2,1:2,A) + ((-1)^(floor(i/2)))*hmat(1:2,1:2,B) + ..
            ((-1)^(i))*hmat(1:2,1:2,C);
            //1 = E11, 2 = E12, 3 = E21, 4 = E22
            if tempMatrix == E11 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/2)))*B;
                EList(x,3) = ((-1)^(i))*C;
                EList(x,5) = i; EList(x,6) = 1; x = x+1;
            end;
            if tempMatrix == E12 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/2)))*B;
                EList(x,3) = ((-1)^(i))*C;
                EList(x,5) = i; EList(x,6) = 2; x = x+1;
            end;
            if tempMatrix == E21 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/2)))*B;
                EList(x,3) = ((-1)^(i))*C;
                EList(x,5) = i; EList(x,6) = 3; x = x+1;
            end;
            if tempMatrix == E22 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/2)))*B;
                EList(x,3) = ((-1)^(i))*C;
                EList(x,5) = i; EList(x,6) = 4; x = x+1;
            end;
            if tempMatrix == (-1)*E11 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/2)+1))*B;
                EList(x,3) = ((-1)^(i+1))*C;
                EList(x,5) = i+4; EList(x,6) = 1; x = x+1;
            end;
            if tempMatrix == (-1)*E12 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/2)+1))*B;
                EList(x,3) = ((-1)^(i+1))*C;
                EList(x,5) = i+4; EList(x,6) = 2; x = x+1;
            end;
            if tempMatrix == (-1)*E21 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/2)+1))*B;
                EList(x,3) = ((-1)^(i+1))*C;
                EList(x,5) = i+4; EList(x,6) = 3; x = x+1;
            end;
            if tempMatrix == (-1)*E22 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/2)+1))*B;
                EList(x,3) = ((-1)^(i+1))*C;
                EList(x,5) = i+4; EList(x,6) = 4; x = x+1;
            end;
        end; //for loop

        //iterate to the next unique set of matrices:
        if C < 16 then
            C = C + 1; //advance C and reset D
        elseif B < 15 then
            B = B + 1; //advance B and reset others
            C = B + 1;
        elseif A < 14 then
            A = A + 1; //advance A and reset others
            B = A + 1;
            C = B + 1;
        end;
    end;
    //72 results (18*4)
endfunction

function[EList] = fourTermSolutions()
    vects = [ 1  1  1  0; //a much smaller set of vectors
              1 -1  0  1];
    m = 0; //counts up the matrices
    hmat = hypermat([2, 2, 16]); //a much smaller set of matrices
    for i = 1:4
        for j = 1:4
            m = m+1;
            tempMatrix = vects(2*i-1:2*i)*vects(2*j-1:2*j)';
            hmat(1:2,1:2,m) = tempMatrix;
        end;
    end;
    E11 = [1,0;0,0];
    E12 = [0,1;0,0];
    E21 = [0,0;1,0];
    E22 = [0,0;0,1];
    x = 1; y = 0;
    A = 1; B = 2; C = 3; D = 4;

    while A < 13
        for i = 0:7 //eight combinations of +/-
            //add and subtract matrices in various combinations
            tempMatrix = hmat(1:2,1:2,A) + ((-1)^(floor(i/4)))*hmat(1:2,1:2,B) + ..
```

```
            ((-1)^(floor(i/2)))*hmat(1:2,1:2,C) + ((-1)^(i))*hmat(1:2,1:2,D);
            //1 through 4 are the four matrices summed together; 5 is the matrix's +/- configuration;
            //6 tells which Eij it is and if it's negative
            //1 = E11, 2 = E12, 3 = E21, 4 = E22
            if tempMatrix == E11 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/4)))*B; EList(x,3) = ((-1)^(floor(i/2)))*C;
                EList(x,4) = ((-1)^(i))*D; EList(x,5) = i; EList(x,6) = 1; x = x+1;
            end;
            if tempMatrix == E12 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/4)))*B; EList(x,3) =..
                 ((-1)^(floor(i/2)))*C;
                EList(x,4) = ((-1)^(i))*D; EList(x,5) = i; EList(x,6) = 2; x = x+1;
            end;
            if tempMatrix == E21 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/4)))*B; EList(x,3) =..
                 ((-1)^(floor(i/2)))*C;
                EList(x,4) = ((-1)^(i))*D; EList(x,5) = i; EList(x,6) = 3; x = x+1;
            end;
            if tempMatrix == E22 then
                EList(x,1) = A; EList(x,2) = ((-1)^(floor(i/4)))*B; EList(x,3) =..
                 ((-1)^(floor(i/2)))*C;
                EList(x,4) = ((-1)^(i))*D; EList(x,5) = i; EList(x,6) = 4; x = x+1;
            end;
            if tempMatrix == (-1)*E11 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/4)+1))*B; EList(x,3) =..
                 ((-1)^(floor(i/2)+1))*C;
                EList(x,4) = ((-1)^(i+1))*D; EList(x,5) = i+8; EList(x,6) = 1; x = x+1;
            end;
            if tempMatrix == (-1)*E12 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/4)+1))*B; EList(x,3) =..
                 ((-1)^(floor(i/2)+1))*C;
                EList(x,4) = ((-1)^(i+1))*D; EList(x,5) = i+8; EList(x,6) = 2; x = x+1;
            end;
            if tempMatrix == (-1)*E21 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/4)+1))*B; EList(x,3) =..
                 ((-1)^(floor(i/2)+1))*C;
                EList(x,4) = ((-1)^(i+1))*D; EList(x,5) = i+8; EList(x,6) = 3; x = x+1;
            end;
            if tempMatrix == (-1)*E22 then
                EList(x,1) = (-1)*A; EList(x,2) = ((-1)^(floor(i/4)+1))*B; EList(x,3) =..
                 ((-1)^(floor(i/2)+1))*C;
                EList(x,4) = ((-1)^(i+1))*D; EList(x,5) = i+8; EList(x,6) = 4; x = x+1;
            end;
        end; //for loop

        //iterate to the next unique set of matrices:
        if D < 16 then //D can be advanced
            D = D + 1; //advance D
        elseif C < 15 then
            C = C + 1; //advance C and reset D
            D = C + 1;
        elseif B < 14 then
            B = B + 1; //advance B and reset others
            C = B + 1;
            D = C + 1;
        elseif A < 13 then
            A = A + 1; //advance A and reset others
            B = A + 1;
            C = B + 1;
            D = C + 1;
        end;
        //y = y + 1 //shows progress of the loop
    end;
    // 352 results (88*4)
endfunction

function[sols] = allTermSolutions()
    EList = oneTermSolutions();
    temp = twoTermSolutions();
    t = size(temp); // t = [16,6]
    e = size(EList);
    EList(e(1)+1:e(1)+t(1),1:6) = temp;
    temp = threeTermSolutions();
    t = size(temp); // t = [72,6]
    e = size(EList);
    EList(e(1)+1:e(1)+t(1),1:6) = temp;
    temp = fourTermSolutions();
    t = size(temp); // t = [352,6]
    e = size(EList);
    EList(e(1)+1:e(1)+t(1),1:6) = temp;
    sols = EList;
endfunction


//Accepts any number of EList rows and returns equivalent Unmatched rows
function[fiveByFive] = EListFormatTranslate(EList)
    s = size(EList); s = s(1); // s = number of rows of EList
    fbfcount = 1;
    for i = 1:s
        currentQuadrant = EList(i,6);
        for j = 1:4
            if EList(i,j) <> 0 then
```

```
                fiveByFive(j,1:2,fbfcount) = [EList(i,j),currentQuadrant]
            end
        end
        fbfcount = fbfcount + 1;
    end
    fiveByFive = EListIndexTranslate(fiveByFive)
endfunction

//Accepts any number of "unmatched" elements, and fixes their indices (1-16 to 1-25)
function[newIndex] = EListIndexTranslate(oldIndex)
    newIndex = abs(oldIndex);
    s = size(newIndex)
    disp(s)
    s2 = size(s)
    disp(s2)
    if s2(2) == 2 then //not a hypermatrix, just a matrix
        for i = 1:s(1)
            select newIndex(i,1)
            case 1 then
                newIndex(i,1) = newIndex(i,1)*sign(oldIndex(i,1))
            case 2 then
                newIndex(i,1) = newIndex(i,1)*sign(oldIndex(i,1))
            case 3 then
                newIndex(i,1) = newIndex(i,1)*sign(oldIndex(i,1))
            case 4 then
                newIndex(i,1) = newIndex(i,1)*sign(oldIndex(i,1))
            case 5 then
                newIndex(i,1) = 6*sign(oldIndex(i,1));
            case 6 then
                newIndex(i,1) = 7*sign(oldIndex(i,1));
            case 7 then
                newIndex(i,1) = 8*sign(oldIndex(i,1));
            case 8 then
                newIndex(i,1) = 9*sign(oldIndex(i,1));
            case 9 then
                newIndex(i,1) = 11*sign(oldIndex(i,1));
            case 10 then
                newIndex(i,1) = 12*sign(oldIndex(i,1));
            case 11 then
                newIndex(i,1) = 13*sign(oldIndex(i,1));
            case 12 then
                newIndex(i,1) = 14*sign(oldIndex(i,1));
            case 13 then
                newIndex(i,1) = 16*sign(oldIndex(i,1));
            case 14 then
                newIndex(i,1) = 17*sign(oldIndex(i,1));
            case 15 then
                newIndex(i,1) = 18*sign(oldIndex(i,1));
            case 16 then
                newIndex(i,1) = 19*sign(oldIndex(i,1));
            end
        end
    elseif s2(2) == 3 then //hypermatrix
        for j = 1:s(3)
            for i = 1:s(1)
                select newIndex(i,1,j)
                case 1 then
                    newIndex(i,1,j) = newIndex(i,1,j)*sign(oldIndex(i,1,j))
                case 2 then
                    newIndex(i,1,j) = newIndex(i,1,j)*sign(oldIndex(i,1,j))
                case 3 then
                    newIndex(i,1,j) = newIndex(i,1,j)*sign(oldIndex(i,1,j))
                case 4 then
                    newIndex(i,1,j) = newIndex(i,1,j)*sign(oldIndex(i,1,j))
                case 5 then
                    newIndex(i,1,j) = 6*sign(oldIndex(i,1,j));
                case 6 then
                    newIndex(i,1,j) = 7*sign(oldIndex(i,1,j));
                case 7 then
                    newIndex(i,1,j) = 8*sign(oldIndex(i,1,j));
                case 8 then
                    newIndex(i,1,j) = 9*sign(oldIndex(i,1,j));
                case 9 then
                    newIndex(i,1,j) = 11*sign(oldIndex(i,1,j));
                case 10 then
                    newIndex(i,1,j) = 12*sign(oldIndex(i,1,j));
                case 11 then
                    newIndex(i,1,j) = 13*sign(oldIndex(i,1,j));
                case 12 then
                    newIndex(i,1,j) = 14*sign(oldIndex(i,1,j));
                case 13 then
                    newIndex(i,1,j) = 16*sign(oldIndex(i,1,j));
                case 14 then
                    newIndex(i,1,j) = 17*sign(oldIndex(i,1,j));
                case 15 then
                    newIndex(i,1,j) = 18*sign(oldIndex(i,1,j));
                case 16 then
                    newIndex(i,1,j) = 19*sign(oldIndex(i,1,j));
                end
            end
        end
    end
```

```
endfunction

//Q is the quadrant of the 20x20 matrix that we are looking for solutions for
function[sols] = allSolutionsForQuadrant(q, fiveByFive)
    f = size(fiveByFive);
    f = f(3); //f = number of matrices in the hypermatrix
    solcount = 1;
    for i = 1:f
        if fiveByFive(1,2,i) == q then
            sols(:,:,solcount) = fiveByFive(:,:,i);
            solcount = solcount + 1;
        end
    end
endfunction

//Receives all 5x5 matrices from a single quadrant and combines them in every possible combination,
//so that GeneratesAllGrids can analyze them
function[allCombos] = allTenByTenCombinations(fiveByFive)
    disp("Calculating all 10x10 varieties... Count up to 112:")
    f = size(fiveByFive);
    f = f(3);
    comboCount = 1;
    for i = 1:f
        disp(i)
        for j = 1:f
            allCombos(1:8,1:2,comboCount) = zeros(8,2)
            entryCount = 1;
            for k = 1:4
                if (fiveByFive(k,1,i) <> 0) then
                    allCombos(entryCount,1,comboCount) = fiveByFive(k,1,i);
                    allCombos(entryCount,2,comboCount) = 1;
                    // the elements from 5x5 grid i go into quadrant 1
                    entryCount = entryCount + 1;
                end
            end
            for k = 1:4
                if (fiveByFive(k,1,j) <> 0) then
                    allCombos(entryCount,1,comboCount) = fiveByFive(k,1,j);
                    allCombos(entryCount,2,comboCount) = 4;
                    // the elements from 5x5 grid j go into quadrant 4
                    entryCount = entryCount + 1;
                end
            end
            if(entryCount <= 6)
                //If the grid has too many elements then it won't be useful, so we overwrite it
                comboCount = comboCount + 1;
            elseif  i == f & j == f then
                //the last one isn't overwritten, so overwrite here.
                allCombos(1:8,1:2,comboCount) = zeros(8,2);
            end
        end
    end
endfunction

//Input: A grid containing four elements, formatted with the 1-16 indices
//example formatting [A,B,C,D]
function[yorn] = TypeThreeSolutionExists(fourElementGrid)
    fourElementGrid = abs(fourElementGrid);
    yorn = %f;
    for i = 1:4
        //disp(i,"i =")
        for j = 1:4
            //disp(j,"j =")
            if (i == j) then //if they are the same element, or are in different quadrants, skip
                continue;
            end
            for k = 1:4
                //disp(k,"k =")
                if (i == k) | (j == k) then //elements must be unique
                    continue;
                else
                    //find out if the three are aligned properly
                    //this goes through redundant sets in different orderings, so we can check a
                    // single ordering without loss of generality
                    if (abs(modulo(fourElementGrid(i),4)) ==..
                        abs(modulo(fourElementGrid(j),4))) & ..
                        (ceil(abs(fourElementGrid(i)/4)) ==..
                         ceil(abs(fourElementGrid(k)/4))) then
                        //if i and j share a column and i and k share a row
                        disp(i,j,k)
                        yorn = %t;
                    end
                end
            end
        end
    end
endfunction

function[w] = allGridsForE(n) // n = 1,2,3, or 4
x = allTermSolutions()
y = EListFormatTranslate(x)
z = allSolutionsForQuadrant(n, y) //four different lists for arguments 1,2,3,4
```

```
w = allTenByTenCombinations(z);
w = w(:,:,1:$-1); //remove the last entry--it's just a bunch of zeros
endfunction

//Takes a 4D hypermatrix and condenses it into a 3D hypermatrix, such that each
// "Slice of the hypermatrix"
// is one of four matrices, each one containing a VERY LARGE number of columns,
//where each column is one solution square
function[processedSols] = processSolutions(solutions)
    s = size(solutions); //usually: s(1) = 4, s(2) = 169, s(3) = 2416, s(4) = 4
    //processedSols = zeros(4,3000?,4)
    for i = 1:s(4)
        solsCount = 1;
        disp(i)
        for j = 1:s(2)
            if (modulo(j,25) == 0) then //progress report
                disp(j)
            end
            for k = 1:s(3)
                //if (solutions(1:4,j,k,i) <> [0;0;0;0])
                if (solutions(1,j,k,i) <> 0)
                    tempSols(1:4,solsCount,i) = solutions(1:4,j,k,i)
                    solsCount = solsCount+1;
                end
            end
        end
        //disp(size(tempSols))
    end
    //processedSols = zeros(tempSols');
    //disp(tempSols');
    for i = 1:4
        processedSols(:,:,i) = tempSols(:,:,i)';
    end
endfunction

function[zeroes] = numberOfZeros(array)
    s = size(array);
    zeroes = 0;
    for i = 1:abs(prod(s))
        if array(i) == 0 then
            zeroes = zeroes + 1;
        end
    end
endfunction

function[newInds] = indexReplace(indices, original, replacement)
    newInds = indices;
    for i = 1:prod(size(indices))
        if abs(indices(i)) == abs(original)) then
            newInds(i) = abs(replacement)*sign(indices(i));
        end
    end
endfunction

//unmatched(:,1) = element index; unmatched(:,2) = element quadrant
function[gridVariant] = findAllGrids(unmatched)
    s = size(unmatched); s = s(1);
    colscore = zeros(s); rowscore = zeros(s);
    finalSize = 16; //the max number of elements in the final "unmatched"
    baseGrid = zeros(finalSize,2);
    baseGrid(1:s,1:2) = unmatched(1:s,1:2);
    sameCol = zeros(2,2,8);
    sameRow = zeros(2,2,8);
    scsize = 1; srsize = 1;
    for i = 1:(s-1)
        if unmatched(i,:) == [0,0] then continue; end
        for j = (i+1):s
            if (abs(modulo(unmatched(i,1),5)) == abs(modulo(unmatched(j,1),5)) & ..
                 (unmatched(i,2) == unmatched(j,2)) & (unmatched(j,:) <> [0,0])) then
                //if the modulo is equal, they are in the same column
                sameCol(1,1:2,scsize) = unmatched(i,:);
                sameCol(2,1:2,scsize) = unmatched(j,:);
                scsize = scsize+1;
            end
            if (abs(ceil(unmatched(i,1)/5)) == abs(ceil(unmatched(j,1)/5)) & ..
                (unmatched(i,2) == unmatched(j,2)) & (unmatched(j,:) <> [0,0])) then
                //if ceilings are equal, they are in the same row
                sameRow(1,1:2,srsize) = unmatched(i,:);
                sameRow(2,1:2,srsize) = unmatched(j,:);
                srsize = srsize+1;
            end
        end
    end

    options = zeros(16,2,1); opGroup = 1;
    //opGroup denotes separate "slices" of the hypermatrix
    for i = 1:s
        if(unmatched(i,2) <> 0) then
            soloMatches = findSoloMatch(unmatched(i,:));
            options(1:4,:,opGroup) = soloMatches;
            opGroup = opGroup + 1;
        end
```

```
    end

    count = 1;
    for i = 1:(s-1)
        for j = i:s
            if (unmatched(i,2) == 1) & (unmatched(j,2) == 4) | ..
                (unmatched(i,2) == 4) & (unmatched(j,2) == 1) then
                temp = findTypeOneMatch([unmatched(i,:);unmatched(j,:)]);
                //ft1dm returns some matrices as a hypermatrix
                t = size(temp); t = t(3) //t = number of hypermatrices
                options(1:16,:,opGroup:opGroup+t-1) = temp(1:16,:,1:t)
            end
        end
    end

    for i = 1:scsize
        if (sameCol(:,:,i) <> [0,0; 0,0]) then
            temp = findTypeTwoMatch(sameCol(:,:,i))
            options(1:8,:,opGroup:opGroup+7) = temp;
            opGroup = opGroup+8;
        end
    end
    for i = 1:srsize
        if (sameRow(:,:,i) <> [0,0; 0,0]) then
            temp = findTypeTwoMatch(sameRow(:,:,i))

            options(1:8,:,opGroup:opGroup+7) = temp;
            opGroup = opGroup+8;
        end
    end

    for i = 1:s
        for j = 1:s
            if (i == j) | unmatched(i,2) <> unmatched(j,2) then
                //if they are the same element, or are in different quadrants, skip
                continue;
            end
            for k = 1:s
                if (i == k) | (j == k) | unmatched(k,2) <> unmatched(i,2) | ..
                    unmatched(k,2) <> unmatched(j,2) then
                    //elements must be unique and in the same quadrant
                    continue;
                else
                    //find out if the three are aligned properly
                    //this goes through redundant sets in different orderings,
                    //so we can check a single ordering without loss of generality
                    if (abs(modulo(unmatched(i,1),5)) == abs(modulo(unmatched(j,1),5))) ..
                        & (ceil(abs(unmatched(i,1)/5)) == ceil(abs(unmatched(k,1)/5))) then
                        for m = 1:s
                            if (m == i) | (m == j) | (m == k) then
                                //must have 4 unique entries
                                continue;
                            elseif (5 - unmatched(m,2) == unmatched(i,2)) then
                                //m must be in the quadrant opposite to i,j,k
                                matchRows(1:4,1:2) = [unmatched(i,:);
                                                      unmatched(j,:);
                                                      unmatched(k,:);
                                                      unmatched(m,:)];
                                //elements to be matched
                                options(1:12,:,opGroup) = findTypeThreeMatch(matchRows);
                                //disp(options(:,:,opGroup))
                                opGroup = opGroup + 1; //iterate to the next group of options
                            end
                        end
                    end
                end
            end
        end
    end

    //find type 4 dual match
    for i = 1:s
        for j = 1:s
            if (unmatched(i,2) <> unmatched(j,2) | i == j) then
                continue;
            end
            for k = 1:s
                if i == k | j == k then
                    continue;
                end
                for m = 1:s
                    if i == m | j == m | k == m then
                        continue;
                    elseif unmatched(i,2) == 1 & unmatched(j,2) == 1 & ..
                        unmatched(k,2) == 4 & unmatched(m,2) == 4 then
                        matchRows(1:4,1:2) = [unmatched(i,:);
                                              unmatched(j,:);
                                              unmatched(k,:);
                                              unmatched(m,:)];
                        t4dmTemp = findTypeFourMatch(matchRows)
                        //if it fails, t4dmTemp == -1
                        if size(size(t4dmTemp)) == [1,3] then
```

```
                        options(1:12,:,opGroup:opGroup+3) = t4dmTemp;
                    end
                end
            end
        end
    end
    gridVariant = hydraSlayer(unmatched, options)
endfunction

function[numberOfSlices] = hsize(grid)
    temp = size(grid);
    numberOfSlices = temp(3);
endfunction

function[numberOfRows] = gsize(grid)
    temp = size(grid);
    numberOfRows = temp(2);
endfunction

//"grid" contains the "unmatched" encoding of a grid, add the "row" to the first
//row of it that is a string of zeros
function[newGrid] = addToFirstEmptyRow(grid, row)
    newGrid = grid;
    //check that row is of the right dimension
    rsize = size(row); //rsize = number of rows to be added
    if rsize(2) <> 2 then
        disp("Error in addToFirstEmptyRow(), row size incorrect.");
        pause
    end
    rsize = rsize(1)
    //check that row will fit (first: calculate how much space remains)
    temp = size(grid); s = temp(1); emptySpaces = 0;

    rowCounter = 1;
    i = 1;
    while ((rowCounter <= rsize) & (i <= s))
        if newGrid(i,1:2) == [0,0] then
            newGrid(i,1:2) = row(rowCounter, 1:2);
            rowCounter = rowCounter + 1;
        end
        i = i + 1;
    end
    if(i == s+1)
        newGrid(i:(i+rsize-rowCounter),1:2) = row(rowCounter:rsize,1:2);
    end
endfunction

//takes one row of Unmatched and returns four rows, containing the original
//square and three new virtual squares
function[soloMatch] = findSoloMatch(unmatchedRow)
    neg = sign(unmatchedRow(1));
    unmatchedRow(1) = abs(unmatchedRow(1));
    soloMatch = zeros(4,2)
    select unmatchedRow(2)
    case 1 then
        soloMatch(1,:) = [neg*unmatchedRow(1),unmatchedRow(2)];
        soloMatch(2,:) = [neg*(ceil(unmatchedRow(1)/5)*5),2];
        soloMatch(3,:) = [neg*(21+modulo(unmatchedRow(1)-1,5)),3];
        soloMatch(4,:) = [neg*25,4];
    case 2 then
        soloMatch(1,:) = [neg*(ceil(unmatchedRow(1)/5)*5),1];
        soloMatch(2,:) = [neg*unmatchedRow(1),unmatchedRow(2)];
        soloMatch(3,:) = [neg*25,3];
        soloMatch(4,:) = [neg*(21+modulo(unmatchedRow(1)-1,5)),4];
    case 3 then
        soloMatch(1,:) = [neg*(21+modulo(unmatchedRow(1)-1,5)),1];
        soloMatch(2,:) = [neg*25,2];
        soloMatch(3,:) = [neg*unmatchedRow(1),unmatchedRow(2)];
        soloMatch(4,:) = [neg*(ceil(unmatchedRow(1)/5)*5),4];
    case 4 then
        soloMatch(1,:) = [neg*25,1];
        soloMatch(2,:) = [neg*(21+modulo(unmatchedRow(1)-1,5)),2];
        soloMatch(3,:) = [neg*(ceil(unmatchedRow(1)/5)*5),3];
        soloMatch(4,:) = [neg*unmatchedRow(1),unmatchedRow(2)];
    else
        soloMatch = zeros(4,2)
    end
endfunction

function[returnedRows] = setNeg(rows, neg)
    if (sign(neg) == 1)
        returnedRows = rows;
    else
        s = size(rows); s = s(1);
        for i = 1:s
            returnedRows(i,1) = rows(i,1)*sign(neg);
            returnedRows(i,2) = rows(i,2);
        end
    end
endfunction
```

```
//Matches the first element in the grid with all the appropriate elements in
//the opposite quadrant in Type 1 dual matches
//and returns an options hypermatrix
//Negatives are always set to "Same" or "Diagonal" arrangement
function[options] = findTypeOneMatch(grid)
    options = zeros(16,2,2);
    grid = sortByQuadrant(grid);
    focus = grid(1,1:2); //Focus is always in Quadrant 1
    n = 1;
    g = size(grid); g = g(1);
    for i = 2:g
        if (grid(i,2) == 4) then //they are in opposite quadrants
            if (sign(focus(1)) == sign(grid(i,1))) then //if both are the same sign
                neg = sign(focus(1)); focus(1) = abs(focus(1));
                tempgrid = abs(grid); //need absolute value for calculations
                //Focus and -1 from VE
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = setNeg(inters(focus, tempgrid(i,:)),-1); //virtual element in Q2
                options(3,:,n) = [modulo(tempgrid(i,1)-1,5)+21,4]; //zero row/column
                options(4,:,n) = setNeg([modulo(focus(1)-1,5)+21,3],-1); //zero row/column
                //### grid(i) and 1 from VE
                options(5,:,n) = tempgrid(i,:); //quadrant 4 element
                options(6,:,n) = inters(focus, tempgrid(i,:)); //virtual element in Q2
                options(7,:,n) = [ceil(focus(1)/5)*5,1]; //zero row/column
                options(8,:,n) = [ceil(tempgrid(i,1)/5)*5,3]; //zero row/column
                options(:,:,n) = setNeg(options(:,:,n),neg); //Reintroduce the negative
                n = n + 1;
                //##
                //##
                // Focus and 1 from VE
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = inters(focus, tempgrid(i,:)); //virtual element in Q2
                options(3,:,n) = [modulo(tempgrid(i,1)-1,5)+21,4]; //zero row/column Q4
                options(4,:,n) = [modulo(focus(1)-1,5)+21,3]; //zero row/column Q3
                //### Grid(i) and -1 from VE
                options(5,:,n) = tempgrid(i,:); //quadrant 4 element
                options(6,:,n) = setNeg(inters(focus, tempgrid(i,:)),-1); //virtual element in Q2
                options(7,:,n) = [ceil(focus(1)/5)*5,1]; //zero row/column Q1
                options(8,:,n) = setNeg([ceil(tempgrid(i,1)/5)*5,3],-1); //zero row/column Q3
                options(:,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
                //Now we also include the "transpose" of this arrangement
                //where the virtual element is in quadrant 3 instead
                //At this point, all T1DMs have the pattern original element,
                //virtual element, zero element, zero element
                //Quadrants 1 and 4 remain the same
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = setNeg(inters(tempgrid(i,:), focus),-1); //virtual element in Q3
                options(3,:,n) = setNeg([ceil(abs(options(2,1,n))/5)*5,4],..
                sign(options(2,1,n)) ); //zero col Q4
                options(4,:,n) = setNeg([ceil(abs(options(1,1,n))/5)*5,2],..
                sign(options(1,1,n)) ); //zero col Q2
                //### grid(i) and 1 from VE
                options(5,:,n) = tempgrid(i,:); //quadrant 4 element
                options(6,:,n) = inters(tempgrid(i,:), focus); //virtual element in Q3
                options(7,:,n) = setNeg([modulo(abs(options(6,1,n)),5)+20, 1],..
                 sign(options(5,1,n))); //zero row Q1
                options(8,:,n) = setNeg([modulo(abs(options(7,1,n)),5)+20, 2],..
                 sign(options(6,1,n))); //zero row Q2
                options(:,:,n) = setNeg(options(:,:,n),neg); //Reintroduce the negative
                n = n + 1;
                //##
                // Focus and 1 from VE
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = inters(tempgrid(i,:), focus); //virtual element in Q3
                options(3,:,n) = setNeg([ceil(abs(options(2,1,n))/5)*5,4],..
                sign(options(2,1,n)) ); //zero col Q4
                options(4,:,n) = setNeg([ceil(abs(options(1,1,n))/5)*5,2],..
                sign(options(1,1,n)) ); //zero col Q2
                //### Grid(i) and -1 from VE
                options(5,:,n) = tempgrid(i,:); //quadrant 4 element
                options(6,:,n) = setNeg(inters(tempgrid(i,:), focus),-1); //virtual element in Q3
                options(7,:,n) = setNeg([modulo(abs(options(6,1,n)),5)+20, 1], sign(options(5,1,n))); //zero row Q1
                options(8,:,n) = setNeg([modulo(abs(options(7,1,n)),5)+20, 2], sign(options(6,1,n))); //zero row Q2
                options(:,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
            else //focus and grid(i) are opposite signs
                neg = sign(focus(1)); focus(1) = abs(focus(1)); //-1*neg = sign of tempgrid(i,1)
                tempgrid = abs(grid); //need absolute value for calculations
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = inters(focus, tempgrid(i,:)); //virtual element in Q2
                options(3,:,n) = [modulo(tempgrid(i,1)-1,5)+21,4]; //zero row/column Q4
                options(4,:,n) = [modulo(focus(1)-1,5)+21,3]; //zero row/column Q3
                options(1:4,:,n) = setNeg(options(1:4,:,n),-1);
                //###
                options(5,:,n) = tempgrid(i,:); //quadrant 4 element
                options(6,:,n) = inters(focus, tempgrid(i,:)); //virtual element in Q2
                options(7,:,n) = [ceil(focus(1)/5)*5,1]; //zero row/column Q1
                options(8,:,n) = [ceil(tempgrid(i,1)/5)*5,3]; //zero row/column Q3
                options(:,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
                //##
```

```
                //##
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = setNeg(inters(focus, tempgrid(i,:)),-1); //virtual element in Q2
                options(3,:,n) = [modulo(tempgrid(i,1)-1,5)+21,4]; //zero row/column Q4
                options(4,:,n) = setNeg([modulo(focus(1)-1,5)+21,3],-1); //zero row/column Q3
                //###
                options(5,:,n) = setNeg(tempgrid(i,:),-1); //quadrant 4 element
                options(6,:,n) = inters(focus, tempgrid(i,:)); //virtual element in Q2
                options(7,:,n) = setNeg([ceil(focus(1)/5)*5,1],-1); //zero row/column Q1
                options(8,:,n) = [ceil(tempgrid(i,1)/5)*5,3]; //zero row/column Q3
                options(:,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
                //Now we also include the "transpose" of this arrangement
                //where the virtual element is in quadrant 3 instead
                //At this point, all T1DMs have the pattern original element
                //virtual element, zero element, zero element
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = inters(tempgrid(i,:), focus); //virtual element in Q3
                options(3,:,n) = [ceil(abs(options(2,1,n))/5)*5*..
                sign(options(2,1,n)),4]; //zero col Q4
                options(4,:,n) = [ceil(abs(options(1,1,n))/5)*5*..
                sign(options(1,1,n)),2]; //zero col Q2
                options(1:4,:,n) = setNeg(options(1:4,:,n),-1);
                //###
                options(5,:,n) = tempgrid(i,:); //quadrant 4 element
                options(6,:,n) = inters(tempgrid(i,:), focus); //virtual element in Q3
                options(7,:,n) = setNeg([modulo(abs(options(6,1,n)),5)+20, 1],..
                 sign(options(5,1,n))); //zero row Q1
                options(8,:,n) = setNeg([modulo(abs(options(7,1,n)),5)+20, 2],..
                 sign(options(6,1,n))); //zero row Q2
                options(:,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
                //##
                options(1,:,n) = focus; //quadrant 1 element
                options(2,:,n) = setNeg(inters(tempgrid(i,:),focus),-1); //virtual element in Q3
                options(3,:,n) = [ceil(abs(options(2,1,n))/5)*5*..
                sign(options(2,1,n)),4]; //zero col Q4
                options(4,:,n) = [ceil(abs(options(1,1,n))/5)*5*..
                sign(options(1,1,n)),2]; //zero col Q2
                //###
                options(5,:,n) = setNeg(tempgrid(i,:),-1); //quadrant 4 element
                options(6,:,n) = inters(tempgrid(i,:), focus); //virtual element in Q3
                options(7,:,n) = setNeg([modulo(abs(options(6,1,n)),5)+20, 1],..
                 sign(options(5,1,n))); //zero row Q1
                options(8,:,n) = setNeg([modulo(abs(options(7,1,n)),5)+20, 2],..
                 sign(options(6,1,n))); //zero row Q2
                options(:,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
            end
        end
    end
    for i = 1:(n-1)
        if(numberOfElements(options(:,:,i)) <> 8)
            disp("Error: incorrect options formed in FindTypeOneMatch")
            disp(options(:,:,i),"incorrect options:")
            disp(grid, "original grid:")
            disp(i, "Option #:")
            pause;
        end
    end
endfunction

//Finds the intersection of two elements (with their indices and quadrants given) A and B
//If A is Q1 and B is Q4, this returns Q2; other way around returns Q3.
//Priority: Same, Diag, Row, Col
function[index] = inters(A, B)
    neg = sign(A(1));
    A = abs(A); B = abs(B);
    if (A(2) == 1 & B(2) == 4) then
        index = [neg*(floor((A(1)-1)/5)*5 + modulo((B(1)-1),5)+1),2];
    elseif (A(2) == 4 & B(2) == 1) then
        index = [neg*(floor((A(1)-1)/5)*5 + modulo((B(1)-1),5)+1),3];
    end
    //Sign(A) is always the sign of the result, because if we have these two diagonals, the
    //final arrangement must be same or rows. index is always on the same row as A, so we use
    //A's sign
    //Frequently, this handles "sanitized" inputs with no negatives--in this case, it will just
    //return the positive anyway
endfunction

//Row is the element that is going to be swapped
//A and B are the elements whose intersection it is on.
//Assumes that A and B are in quadrants 1 and 4 (not necessarily respectively)
//and that row is in quadrant 2 or 3
//I don't know that this is actually useful
function[swapRow] = swapInters(row,A,B)
    if (row(2) == 2) then
        newQuadrant = 3;
    elseif (row(2) == 3) then
        newQuadrant = 2;
    end
    firstInters = inters(A,B);
```

```
    secondInters = inters(B,A);
    if abs(firstInters(1)) == abs(row(1)) then
        swapRow = [secondInters(1), newQuadrant];
    elseif abs(secondInters(1)) == abs(row(1)) then
        swapRow = [firstInters(1), newQuadrant];
    else
        swapRow = 999999; //error
    end
endfunction

// Type 2:
//Two elements in the same quadrant on the same row/col;
//create a virtual element along that row/col in the oter quadrant along it.
//Match each element with this virtual element, then both with the opposite zero row/col
//this assumes that elements do not count each other for row score--
//each pair only occurs once, when the first element counts the second, not the other way around
//rows contains the two elements that share a row/col and a quadrant, though we double check this
//Shrunk to fit paper copies, comments may refer to lines above themselves
function[options] = findTypeTwoMatch(rows)
    if (rows(1,2) <> rows(2,2)) then
        disp("error: element quadrants do not match")
        pause
    end
    if (rows(1,:) == rows(2,:)) then
        disp("error: two identical elements")
        pause
    end
    if rows(1,2) == 1 then
        vquad = 2;
    elseif rows(1,2) == 4 then
        vquad = 3;
    else
        disp("Error: wrong quadrant in unmatched element")
        pause
    end
    n = 1;
    if (ceil(rows(1,1)/5) == ceil(rows(1,1)/5)) then //elements are in the same row
        for i = 1:4
            if (sign(rows(1,1)) == sign(rows(2,1))) then //signs are the same
                //signs are the same (assume both are positive, setNeg handles it)
                neg = sign(rows(1,1)); temprows = abs(rows);
                //rows(1) and -1 from VE
                options(1,:,n) = temprows(1,:) //original element
                options(2,:,n) = setNeg([floor((temprows(1,1)-1)/5)+i, vquad], -1)
                //virtual element
                options(3,:,n) = [20+i, 5-temprows(1,2)] //opposite quadrant zero row
                options(4,:,n) = setNeg([21+modulo(temprows(1,1)-1,5), 5-vquad], -1)
                //opposite to virtual element, zero row
                //### rows(2) and 1 from VE
                options(5,:,n) = temprows(2,:) //original element
                options(6,:,n) = [floor((temprows(1,1)-1)/5)+i, vquad] //virtual element
                options(7,:,n) = [20+i,5-temprows(1,2)]
                //opposite quadrant zero row
                options(8,:,n) = [21+modulo(temprows(2,1)-1,5), 5-vquad]
                //opposite to virtual element, zero row
                options(1:8,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
                //##
                //##
                //Rows(1) and 1 from VE
                options(1,:,n) = temprows(1,:);
                //original element
                options(2,:,n) = [floor((temprows(1,1)-1)/5)+i, vquad];
                //virtual element
                options(3,:,n) = [20+i, 5-temprows(1,2)];
                //opposite quadrant zero row
                options(4,:,n) = [21+modulo(temprows(1,1)-1,5), 5-vquad];
                //opposite to virtual element, zero row
                //### rows(2) and -1 from VE
                options(5,:,n) = temprows(2,:); //original element
                options(6,:,n) = setNeg([floor((temprows(1,1)-1)/5)+i, vquad], -1);
                //virtual element
                options(7,:,n) = [20+i,5-temprows(1,2)]; //opposite quadrant zero row
                options(8,:,n) = setNeg([21+modulo(temprows(2,1)-1,5), 5-vquad], -1);
                //opposite to virtual element, zero row
                options(1:8,:,n) = setNeg(options(:,:,n),neg);
                n = n + 1;
            else //signs are different
                neg = sign(rows(1,1));
                temprows = abs(rows);
                //signs are opposite (assume rows(1,1) is positive, setNeg handles it)
                //rows(1) and -1 from VE (1 neg, 1 pos)
                options(1,:,n) = temprows(1,:); //original element
                options(2,:,n) = setNeg([floor((temprows(1,1)-1)/5)+i, vquad], -1);
                //virtual element
                options(3,:,n) = [20+i, 5-temprows(1,2)]; //opposite quadrant zero row
                options(4,:,n) = setNeg([21+modulo(temprows(1,1)-1,5), 5-vquad], -1);
                //opposite to virtual element, zero row
                //### rows(2) and 1 from VE (1 neg, 1 pos)
                options(5,:,n) = setNeg(temprows(2,:), -1); //original element
                options(6,:,n) = [floor((temprows(1,1)-1)/5)+i, vquad]; //virtual element
                options(7,:,n) = setNeg([20+i,5-temprows(1,2)], -1); //opposite quadrant zero row
```

```
            options(8,:,n) = [21+modulo(temprows(2,1)-1,5), 5-vquad];
            //opposite to virtual element, zero row
            options(1:8,:,n) = setNeg(options(1:8,:,n), neg);
            n = n + 1;
            //##
            //##
            //Rows(1) and 1 from VE (both positive)
            options(1,:,n) = temprows(1,:) //original element
            options(2,:,n) = [floor((temprows(1,1)-1)/5)+i, vquad]; //virtual element
            options(3,:,n) = [20+i, 5-temprows(1,2)]; //opposite quadrant zero row
            options(4,:,n) = [21+modulo(temprows(1,1)-1,5), 5-vquad];
            //opposite to virtual element, zero row
            //### rows(2) and -1 from VE (both negative)
            options(5,:,n) = setNeg(temprows(2,:), -1); //original element
            options(6,:,n) = setNeg([floor((temprows(1,1)-1)/5)+i, vquad], -1);
            //virtual element
            options(7,:,n) = setNeg([20+i,5-temprows(1,2)], -1);
            //opposite quadrant zero row
            options(8,:,n) = setNeg([21+modulo(temprows(2,1)-1,5), 5-vquad], -1);
            //opposite to virtual element, zero row
            options(1:8,:,n) = setNeg(options(:,:,n), neg);
            n = n + 1;
        end
    end
elseif (modulo(rows(1,1)-1,5) == modulo(rows(2,1)-1,5)) then //elements are in the same column
    for i = 1:4
        if (sign(rows(1,1)) == sign(rows(2,1))) then
            //same signs
            neg = sign(rows(1,1)); temprows = abs(rows);
            //always assume element (1,1) is positive
            // (+) rows(1) and -1 from VE
            options(1,:,n) = temprows(1,1); //original element
            options(2,:,n) = setNeg([ceil(temprows(1,1)/5)-4, vquad],-1); //virtual element
            options(3,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(4,:,n) = setNeg([modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad],-1);
            //zero entry
            //### (+) rows(2) and 1 from VE
            options(5,:,n) = temprows(2,1); //original element
            options(6,:,n) = [ceil(temprows(2,1)/5)-4, vquad]; //virtual element
            options(7,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(8,:,n) = [modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad] //zero entry, opposite virtual
            options(1:8,:,n) = setNeg(options(:,:,n),neg);
            //we assumed (1,1) was positive, this fixes it
            n = n + 1;
            //##
            //##
            //(+)Rows(1) and 1 from VE
            options(1,:,n) = temprows(1,1); //original element
            options(2,:,n) = [ceil(temprows(1,1)/5)-4, vquad]; //virtual element
            options(3,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(4,:,n) = [modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad]
            //zero entry, opp virtual
            //### (+)rows(2) and -1 from VE
            options(5,:,n) = temprows(2,1); //original element
            options(6,:,n) = setNeg([ceil(temprows(2,1)/5)-4, vquad], -1); //virtual element
            options(7,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(8,:,n) = setNeg([modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad], -1)
            //zero entry
            options(1:8,:,n) = setNeg(options(:,:,n),neg);
            n = n + 1;
        else
            //different signs
            neg = sign(rows(1,1)); temprows = abs(rows);
            //rows(1) and -1 from VE
            options(1,:,n) = temprows(1,1); //original element
            options(2,:,n) = [ceil(temprows(1,1)/5)-4, vquad]; //virtual element
            options(3,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(4,:,n) = [modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad]
            //zero entry, opp virtual
            //### (-)rows(2) and 1 from VE
            options(5,:,n) = temprows(2,1); //original element
            options(6,:,n) = [ceil(temprows(2,1)/5)-4, vquad]; //virtual element
            options(7,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(8,:,n) = [modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad]
            //zero entry, opp virtual
            options(:,:,n) = setNeg(options(:,:,n),neg);
            n = n + 1;
            //##
            //##
            //Rows(1) and 1 from VE
            options(1,:,n) = temprows(1,1); //original element
            options(2,:,n) = [ceil(temprows(1,1)/5)-4, vquad]; //virtual element
            options(3,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(4,:,n) = [modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad] //zero entry, opp virtual
            //### rows(2) and -1 from VE
            options(5,:,n) = temprows(2,1); //original element
            options(6,:,n) = [ceil(temprows(2,1)/5)-4, vquad]; //virtual element
            options(7,:,n) = [(i*5)-4, 5-temprows(1,2)]; //zero entry, opposite original
            options(8,:,n) = [modulo(temprows(1,1)-1,5)+1 + (i-1)*5, 5 - vquad] //zero entry, opp virtual
            options(1:8,:,n) = setNeg(options(:,:,n),neg);
            n = n + 1;
        end
```

```
        end
    end
endfunction

//Three elements in one quadrant, one the fourth element, in the opposite quadrant
//Now we really need to pay attention to the negatives,
//since one solution square will have nothing in the zero row
function[options] = findTypeThreeMatch(rows)
    //rows contains four elements, three in one quadrant, one in the opposite quadrant
    //the third element in the quadrant is at the intersection of the other two
    rows = t3dmReorder(rows);
    //rows(1,:) = element in its own quadrant
    //rows(2,:) = element at intersection of 3 and 4
    //rows(3,:) = element in same row as 2
    //rows(4,:) = element in same column as 2
    mainQ = rows(2,2); //quadrant that contains 3 elements
    oppQ = 5 - mainQ; //quadrant that contains 1 element
    if mainQ == 1 then
        horizQ = 2;//quadrant horizontal to the mainQ
    elseif mainQ == 4 then
        horizQ = 3;
    else
        disp("Error: screwy quadrants in type 3 match")
        disp(rows(2,2))
        pause
    end
    vertQ = 5 - horizQ; //quadrant vertical to the mainQ
    neg = sign(rows(1,1));
    hneg = sign(rows(3,1));
    vneg = sign(rows(4,1));
    oRows = rows; //original rows; some of these operations require knowledge of the original signs
    absrows = abs(rows); //negatives are otherwise recorded separately
    if (sign(rows(1,1)) == sign(rows(2,1))) then
        //Same signs on the corners, assume they are positive
        options(1, :) = absrows(1,:);
        options(2, :) = inters(absrows(1,:), absrows(2,:));
        options(3, :) = inters(absrows(2,:), absrows(1,:));
        options(4, :) = absrows(2,:);
        options(1:4, :) = setNeg(options(1:4,:), neg)
        //Second solution square: element #3, the virtual element on the horizontal,
        //and two zero-row entries on the vertical
        options(5, :) = setNeg(absrows(3,:), hneg);
        options(6, :) = setNeg([5*floor(absrows(3,1)/5) + modulo(absrows(1,1)-1,5)+1, horizQ], -1*neg);
        options(7, :) = setNeg([20 + modulo(absrows(1,1)-1,5)+1, oppQ], hneg);
        options(8, :) = setNeg([20 + modulo(absrows(1,1)-1,5)+1, vertQ], -1*neg);
        //Third solution square: element #4, the virtual element on the vertical,
        //and two zero-column entries on the horizontal
        options(9, :) = setNeg(absrows(4,:), vneg);
        options(10,:) = setNeg([5*floor(absrows(1,1)/5) + modulo(absrows(4,1)-1,5)+1, vertQ], -1*neg);
        options(11,:) = setNeg([5*ceil(absrows(4,1)/5), horizQ], vneg);
        options(12,:) = setNeg([5*ceil(absrows(1,1)/5), oppQ], -1*neg);
    else//opposite signs on corners, assume element 1 is positive
        //we want elements on the same row to have the same sign
        options(1, :) = absrows(1,:);
        options(2, :) = inters(rows(1,:), rows(2,:)); //use of rows is necessary,
        //as we don't know which element is in which quadrant
        options(3, :) = inters(rows(2,:), rows(1,:));
        options(4, :) = setNeg(absrows(2,:),-1);
        options(1:4, :) = setNeg(options(1:4,:), neg)
        //fix negative signs
        if options(2,2) == 2 then
            options(2,1) = options(2,1)*(-1);
        elseif options(3,2) == 2 then
            options(3,1) = options(3,1)*(-1);
        end
        //Need to find out which one is horizontal to the negative vertQ
        if (options(2,2) == vertQ) then
            options(2,1) = (-1)*options(2,1);
        else
            options(3,1) = (-1)*options(3,1);
        end
        //Second solution square: element #3,
        //the virtual element on the horizontal, and two zero-row entries on the vertical
        options(5, :) = setNeg(absrows(3,:), hneg); //element#3
        options(6, :) = setNeg([5*floor(absrows(3,1)/5) + modulo(absrows(1,1)-1,5)+1, horizQ], -1);
        //virtual element
        options(7, :) = setNeg([20 + modulo(absrows(1,1)-1,5)+1, oppQ], hneg);
        options(8, :) = setNeg([20 + modulo(absrows(1,1)-1,5)+1, vertQ], -1);
        //Third solution square: element #4, the virtual element on the vertical,
        // and two zero-column entries on the horizontal
        options(9, :) = setNeg(absrows(4,:), vneg);
        options(10,:) = [5*floor(absrows(1,1)/5) + modulo(absrows(4,1)-1,5)+1, vertQ];
        options(11,:) = setNeg([5*ceil(absrows(4,1)/5), horizQ], vneg);
        options(12,:) = [5*ceil(absrows(1,1)/5), oppQ];
    end
endfunction


//T3DM helper function, gets the array in the right order
function[orderedRows] = t3dmReorder(rows)
//rows contains four elements. This function reorders them such that the
//first element is in its own quadrant,
```

```
//the second element is the intersection, and the last two are the one sharing a
// row and column with the intersection, respectively.
    if rows(1,2) + rows(2,2) + rows(3,2) > 8 then
        //if more than one of these are 4, three are in quadrant 4. Else, three are in quadrant 1.
        mainQuad = 4;
    else
        mainQuad = 1;
    end
    for i = 1:4
        if rows(i,2) == 5 - mainQuad then //if this element is the one alone in its quadrant
            orderedRows(1,:) = rows(i,:);
            rows(i,:) = rows(4,:);
            rows(4,:) = [0,0]; //swap the fourth element for the deleted element
        end
    end
    //now only three elements remain, all in the same quadrant
    absrows = abs(rows);
    for i = 1:3
        x = i; y = modulo(i,3)+1; z = modulo(i+1,3)+1;
        //x, y, and z are always three different numbers from 1 to 3
        if modulo(absrows(x,1),5) <> modulo(absrows(y,1),5) & ..
            modulo(absrows(x,1),5) <> modulo(absrows(z,1),5) then
            //x shares a row with the intersection
            orderedRows(3,:) = rows(x,:);
        elseif ceil(absrows(x,1)/5) <> ceil(absrows(y,1)/5) & ..
            ceil(absrows(x,1)/5) <> ceil(absrows(z,1)/5) then
            //x shares a column with the intersection
            orderedRows(4,:) = rows(x,:);
        else
            orderedRows(2,:) = rows(x,:);
            //if x is not in its own row or own column then it must be the intersection
        end
    end
endfunction

//Four elements, two in each quadrant. Either each pair shares a row or each pair shares a column
//Does not appear in the Strassen algorithm but does appear in other fast matrix mult algorithms
function[options] = findTypeFourMatch(rows)
    //First: validate the input
    rows = sortByQuadrant(rows);
    valid = %f;
    absrows = abs(rows)
    n = 1;
    //Vertical:
    if (modulo(absrows(1,1),5) == modulo(absrows(2,1),5) & ..
        modulo(absrows(3,1),5) == modulo(absrows(4,1),5)) then
        valid = %t;
        n = 1;
        for i = 1:2
            for j = 3:4
                //Four different possible locations for the virtual elements--
                //for loops determine which two elements are "paired together"
                //NOTE for negatives: always makes "same" arrangement when possible,
                //otherwise "rows" arrangement
                //First solution square, contains the paired elements:
                options(1, :, n) = rows(i,:) //Original Element in Q1
                options(2, :, n) = [(floor((absrows(i,1)-1)/5)*5 + ..
                modulo(absrows(j,1)-1,5) + 1)*sign(rows(i,1)), 2];//virtual element in Q2
                options(3, :, n) = [(floor((absrows(j,1)-1)/5)*5 + ..
                modulo(absrows(i,1)-1,5) + 1)*sign(rows(j,1)), 3];//virtual element in Q3
                options(4, :, n) = rows(j,:) //Original element in Q4
                //Second solution square: contains unpaired element from Q1
                options(5, :, n) = rows(modulo(i,2)+1,:) //The other Original Element in Q1
                options(6, :, n) = [ceil(abs(options(5,1,n))/5)*5*sign(options(5,1,n)),2]; //Zero-col in Q2
                options(7, :, n) = [-1*options(3,1,n),options(3,2,n)];
                //virtual element in Q3, negative of the other one
                options(8, :, n) = [ceil(abs(options(7,1,n))/5)*5*sign(options(7,1,n)),4]; //Zero-col in Q4
                //Third solution square: contains unpaired element from Q4
                options(9, :, n) = [ceil(ceil(floor((absrows(i,1)-1)/5)*5 + ..
                modulo(absrows(j,1)-1,5) + 1)/5)*5*sign(options(2,1,n))*(-1),1] // zero-col element in Q1
                options(10,:, n) = [-1*options(2,1,n),2]; //virtual element
                options(11,:, n) = [ceil(absrows(modulo(j,2)+3,1)/5)*5*sign(rows(modulo(j,2)+3,1)),3]
                // zero-column element in Q3
                options(12,:, n) = rows(modulo(j,2)+3,:) //original element in Q4--if j = 3, this = 4
                n = n + 1;
            end
        end
        //Horizontal:
    elseif (ceil(absrows(1,1)/5) == ceil(absrows(2,1)/5) & ..
            ceil(absrows(3,1)/5) == ceil(absrows(4,1)/5)) then
        valid = %t;
        n = 1;
        for i = 1:2
            for j = 3:4 //Four different possible locations for the virtual elements--
                //for loops determine which two elements are "paired together"
                //NOTE for negatives: always makes "same" arrangement when
                //possible, otherwise "rows" arrangement
                //First solution square, contains the paired elements
                options(1, :, n) = rows(i,:) //Original Element in Q1
                options(2, :, n) = [(floor((absrows(i,1)-1)/5)*5 + ..
                modulo(absrows(j,1)-1,5)+1)*sign(rows(i,1)), 2];//virtual element in Q2
                options(3, :, n) = [(floor((absrows(j,1)-1)/5)*5 + ..
```

```
                modulo(absrows(i,1)-1,5)+1)*sign(rows(j,1)), 3];//virtual element in Q3
                options(4, :, n) = rows(j,:) //Original element in Q4
                //Second solution square: contains unpaired element from Q1
                options(5, :, n) = rows(modulo(i,2)+1,:) //Original Element in Q1
                options(6, :, n) = [-1*options(2,1,n),options(2,2,n)];//virtual element in Q2
                options(7, :, n) = [(modulo(abs(options(5,1,n))-1,5)+21)*..
                sign(options(5,1,n)),3]; //Zero-row element in Q3
                options(8, :, n) = [(modulo(abs(options(6,1,n))-1,5)+21)*..
                sign(options(6,1,n)),4]; //Zero-row element in Q4
                //Third solution square: contains unpaired element from Q4
                options(9, :, n) = [(modulo(floor((absrows(j,1)-1)/5)*5 +..
                 modulo(absrows(i,1)-1,5),5) +21)*sign(options(3,1,n))*(-1), 1] // zero-row element in Q1
                options(10,:, n) = [(modulo(absrows(modulo(j,2)+3,1)-1,5)+21)*..
                sign(rows(modulo(j,2)+3,1)),2] // zero-row element in Q2
                options(11,:, n) = [-1*options(3,1,n),options(3,2,n)];
                options(12,:, n) = rows(modulo(j,2)+3,:) //if j = 3, this = 4
                n = n + 1;
            end
        end
    end

    for i = 1:(n-1)
        if numberOfElements(options(:,:,i)) <> 12 then
            valid = %f;
        end
    end

    if ~valid then
        options = -1;
    end
endfunction

function[viableGridsExist] = hasViableGrids(options, originals)
    maxNumOptions = 0;
    o = size(options);
    o = o(3); //number of hypermatrices
    for i = 1:o
        if numberOfElements(options(:,:,i)) > maxNumOptions then
            maxNumOptions = numberOfElements(options(:,:,i));
        end
    end
    if (numberOfElements(originals) == 5) & (maxNumOptions == 8) then
        viableGridsExist = %f;
    else
        viableGridsExist = %t;
    end
endfunction

//Hydraslayer multiplies the grids and makes makes a
//variant of each initial grid with each of the options,
//essentially multiplying the number of grids by the number of options,
//with the exception of discounted, invalid grids.
//Named because it solves a problem that seemed to create /
//more problems whenever solutions were presented
function[indexGrids] = hydraSlayer(originals, options)
    //allGrids = options; //sometimes, an "option" is actually an entire solution,
    //so we start with all the option groups as potential solution grids
    viableGrids = zeros(16,2,1);
    o = size(options); o = o(3); //o = number of sets of options

    //Determine which quadrant of originals has the most elements, and how many that is
    qcount = [0,0,0,0]';
    ori = numberOfElements(originals);
    //calculate how many entries a grid should have
    for i = 1:ori
        qcount(originals(i,2)) = qcount(originals(i,2)) + 1;
    end
    elementsPerQuadrant = max(qcount); // max(qcount);
    if elementsPerQuadrant < 2 then
        elementsPerQuadrant = 2;
        //if there is one element in each quadrant, we allow two solution squares
    end
    n = elementsPerQuadrant

    if ~hasViableGrids(options, originals) then
        viableGrids = 0;
    else
        temp = allOptions(options, originals);
        if (temp(1,1,1) <> -99) //error code
            viableGrids = allOptions(options, originals);
        end
    end

    n = size(size(viableGrids));
    if (n(2) == 2)
        n = 1;
    else
        n = size(viableGrids); n = n(3);
    end
    for k = 1:n
        indexGrids(1:4,k) = handleIndexing(viableGrids(:,:,k));
        if viableGrids(1,1,1) <> 0 then
```

```
            if indexGrids(4,k) == 0 & viableGrids(13,1,k) <> 0 then
                disp("Hydraslayer: Error in indexing discovered")
                pause;
            end
        end
    end
endfunction

function[allGrids] = allOptions(options, originals)
    allGrids = zeros(16,2,2) //to form it as a hypermatrix
    allGrids(1,1,1) = -99; // sometimes, there are no viable grids so this is the error code
    s = size(options); s = s(3);
    gridCount = 1;
    grid1 = zeros(16,2,1);
    for i = 1:s
        grid1 = options(:,:,i);
        if containsOriginals(grid1, originals) then
            grid1 = grid1(1:16,:)
            allGrids(1:16,:,gridCount) = grid1(1:16,:);
            gridCount = gridCount + 1;
            continue;
        end
        for j = (i+1):s
            grid2 = addToGrid(grid1, options(:,:,j))
            if duplicatesInList(grid2) | numberOfElements(grid2) > 16 then
                continue;
            elseif (containsOriginals(grid2, originals)) then
                grid2 = grid2(1:16,:)
                allGrids(:,:,gridCount) = grid2;
                gridCount = gridCount + 1;
                continue;
            end
            for k = (j+1):s
                grid3 = addToGrid(grid2, options(:,:,k))
                if duplicatesInList(grid3) | numberOfElements(grid3) > 16 then
                    continue;
                elseif (containsOriginals(grid3, originals)) then
                    grid3 = grid3(1:16,:)
                    allGrids(:,:,gridCount) = grid3;
                    gridCount = gridCount + 1;
                    continue;
                end
                for m = (k+1):s
                    grid4 = addToGrid(grid3, options(:,:,m))
                    if duplicatesInList(grid4) | numberOfElements(grid4) > 16 then
                        continue;
                    elseif (containsOriginals(grid4, originals)) then
                        grid4 = grid4(1:16,:)
                        allGrids(:,:,gridCount) = grid4;
                        gridCount = gridCount + 1;
                        continue;
                    end
                end
            end
        end
    end
endfunction

//Adds rows of a 2-column array to the bottom of another array
function[addedGrid] = addToGrid(grid, addition)
    g = size(grid); g = g(1);
    a = size(addition); a = a(1);
    addedGrid = grid;
    acount = 1;
    for i = 1:g
        if (addedGrid(i,:) == [0,0]) & (addition(acount,:) <> [0,0]) then
            addedGrid(i,:) = addition(acount,:)
            acount = acount + 1;
        end
    end
    bcount = acount;
    for i = (g + 1):(g + a - acount - 1)
        addedGrid(i,:) =  addition(bcount,:)
        bcount = bcount + 1;
    end
endfunction

//Specifically for counting the number of elements present in a matrix
//(as they are frequently padded with zeros, making the size function inaccurate)
function[trueSize] = numberOfElements(grid)
    trueSize = 0;
    s = size(grid); s = s(1);
    for i = 1:s
        if grid(i,:) <> [0,0] then
            trueSize = trueSize + 1;
        end
    end
endfunction

//Tests that each original element is present exactly once
function[originalsExistOnce] = containsOriginals(grid, originals)
    originalsExistOnce = %t;
```

```
    g = size(grid); g = g(1);
    o = size(originals); o = o(1);
    present = zeros(originals(:,1))
    //present counts how many of each original element are present
    for i = 1:g
        for j = 1:o
            if (grid(i,:) == originals(j,:)) then
                present(j) = present(j) + 1;
            end
        end
    end
    for i = 1:o
        if (present(i) <> 1) & originals(i,:) <> [0,0] then
            originalsExistOnce = %f;
        end
    end
endfunction

//Takes two lists and checks if any entries are the same
function[duplicatesExist] = containsDuplicates(grid, options)
    duplicatesExist = %f;
    g = size(grid); g = g(1);
    o = size(options); o = o(1);
    for i = 1:g
        for j = 1:o
            //Does not count duplicates appearing in zero-rows or columns
            if ((grid(i,:) == options(j,:)) & (modulo(grid(i,1),5) <> 0) & ..
                (grid(i,1) < 20) & (grid(i,:) <> [0,0])) then
                duplicatesExist = %t
                break;
            end
        end
        if (duplicatesExist) then break; end;
    end
endfunction

function[duplicatesExist] = duplicatesInList(grid)
    duplicatesExist = %f;
    g = size(grid); g = g(1);
    for i = 1:g
        for j = (i+1):g
            if (grid(i,:) == grid(j,:) & (modulo(grid(i,1),5) <> 0) & ..
                (grid(i,1) < 20) & (grid(i,1) > -20)) then
                duplicatesExist = %t
                break;
            end
        end
        if (duplicatesExist) then break; end
    end
endfunction


//Takes a list of entries in unmatched format and "indexifies" them as up to 4 indices
//Since the indexing function uses a different formatting of data
function[indices] = handleIndexing(grid)
    indices = zeros(4,1)
    s = numberOfElements(grid);
    //disp(grid, "The grid:")
    //disp(s, "grid contains this many elements:")
    if(modulo(s,4) <> 0) then
        disp("Error: incorrect number of entries")
        pause
    end
    for i = 1:4:s
        //disp(i, "i = ")
        holder = [0,0;0,0];
        for j = 0:3
            //disp(j, "j = ")
            select grid(i+j,2)
            case 1 then
                holder(1,1) = grid(i+j,1);
            case 2 then
                holder(1,2) = grid(i+j,1);
            case 3 then
                holder(2,1) = grid(i+j,1);
            case 4 then
                holder(2,2) = grid(i+j,1);
            end
        end
        if (holder(1,1) == 0 | holder(1,2) == 0 | holder(2,1) == 0 | holder(2,2) == 0)
            disp("Error: improper arrangement of grid elements:")
            disp(grid(i:i+3,:))
            pause
        end
        //indices(:,:,ceil(i/4)) = indexify(holder);
        indices(ceil(i/4)) = indexify(holder);
    end
    //disp(indices, "handleIndexing returns")
endfunction

//Does the same thing as handleIndexing, but with indexify2
function[indices] = handleIndexing2(grid)
```

```
    indices = zeros(4,1)
    s = numberOfElements(grid);
    //disp(grid, "The grid:")
    //disp(s, "grid contains this many elements:")
    if(modulo(s,4) <> 0) then
        disp("Error: incorrect number of entries")
        pause
    end
    for i = 1:4:s
        //disp(i, "i = ")
        holder = [0,0;0,0];
        for j = 0:3
            //disp(j, "j = ")
            select grid(i+j,2)
            case 1 then
                holder(1,1) = grid(i+j,1);
            case 2 then
                holder(1,2) = grid(i+j,1);
            case 3 then
                holder(2,1) = grid(i+j,1);
            case 4 then
                holder(2,2) = grid(i+j,1);
            end
        end
        if (holder(1,1) == 0 | holder(1,2) == 0 | holder(2,1) == 0 | holder(2,2) == 0)
            disp("Error: improper arrangement of grid elements:")
            disp(grid(i:i+3,:))
            pause
        end
        //indices(:,:,ceil(i/4)) = indexify(holder);
        indices(ceil(i/4)) = indexify2(holder);
    end
    //disp(indices, "handleIndexing returns")
endfunction

//Sorts four entries of unmatched in descending order of quadrant and automatically removes zero-rows
//(Quadrant is in column 2 of the matrix)
function[sorted] = sortByQuadrant(unmatched)
    s = size(unmatched); //s(1) = number of rows
    j = 1;
    for k = 1:4
        for i = 1:s(1)
            if(unmatched(i,2) == k) then
                sorted(j,:) = unmatched(i,:);
                j = j + 1;
            end
        end
    end
endfunction

function[index] = indexify(solutionRectangle)
    negatives = [0, 0; 0, 0;];
    negativeIndex = 0;
    for i = 1:2
        for j = 1:2
            if (solutionRectangle(i,j) > 0) then
                negatives(i,j) = 1;
            elseif (solutionRectangle(i,j) < 0) then
                negatives(i,j) = -1;
            end;
        end;
    end;
    select negatives
    case [1, 1; 1, 1] then
        negativeIndex = 1;
    case [-1, -1; -1, -1] then
        negativeIndex = -1;
    case [1, -1; 1, -1] then
        negativeIndex = 2;
    case [-1, 1; -1, 1] then
        negativeIndex = -2;
    case [1, 1; -1, -1] then
        negativeIndex = 3;
    case [-1, -1; 1, 1] then
        negativeIndex = -3;
    case [1, -1; -1, 1] then
        negativeIndex = 4;
    case [-1, 1; 1, -1] then
        negativeIndex = -4;
    else
        negativeIndex = 0;
    end;
    index = sign(negativeIndex)*(abs((solutionRectangle(1,1))*25) + ..
    abs(solutionRectangle(2,2)) + (abs(negativeIndex))*(651));
endfunction

function[solutionRectangle] = deindexify(index)
    negativeIndex = fix(index/651); //floor does not interact as we want with negative numbers
    select abs(negativeIndex)
    case 1 then
        negatives = sign(negativeIndex)*[1, 1; 1, 1];
    case 2 then
```

```
        negatives = sign(negativeIndex)*[1, -1; 1, -1];
    case 3 then
        negatives = sign(negativeIndex)*[1, 1; -1, -1];
    case 4 then
        negatives = sign(negativeIndex)*[1, -1; -1, 1];
    else
        negatives = 0;
    end;
    index = abs(modulo(index,651)); //remove the negative component of the index
    //We need 1 and 4 to calculate indices 2 and 3, we do them first

    ind1 = (fix((index-1)/25));
    ind4 = (modulo((index-1),25)+1);
    ind2 = (modulo((ind4-1),5)+1 + (fix((ind1-1)/5))*5);
    ind3 = (modulo((ind1-1),5)+1 + (fix((ind4-1)/5))*5);
    solutionRectangle = [ind1, ind2; ind3 ,ind4].*negatives;
    //elementwise multiplication to "apply" proper negatives
endfunction


//Runs in 30 minutes (or your algorithm is free)
function[algorithms] = matchSolutions2(solutions)
    s = size(solutions); s = s(1);
    algCounter = 1; alg = 0;
    //brute force algorithm
    //get number of solution sets, these will be very large
    //sols = zeros(7600)
    tempSol = zeros(4,4);
    //If the first three solution sets have more than 7 unique solutions then
    //all combinations involving them can be skipped
    disp(s(1))
    timer()
    savedComparisons = 0;
    completedComparisons = 0;
    for i = 1:s
        disp(i,"i=")
        tempSol = zeros(4,4);
        //The size checks at each loop will interrupt valid grids without this reset
        if solutions(i,1,1) == 0 then
            continue;
        else
            tempSol(1,1:4) = solutions(i,1:4,1)
        end
        for j = 1:s
            if modulo(j,200) == 0 then
                disp(j)
            end
            tempSol(2,1:4) = [0,0,0,0]; //(this is probably redundant)
            tempSol(3,1:4) = [0,0,0,0]; //Reset
            tempSol(4,1:4) = [0,0,0,0];
            if solutions(j,1,2) == 0 then
                continue;
            else
                tempSol(2,1:4) = solutions(j,1:4,2)
            end
            n = size(unique(abs(tempSol))); n = n(1);
            //n = number of unique solutions
            if (n > 8) then
                //Obviously, if we already have 8 different indices,
                //we can't add more and end up with fewer than 8
                savedComparisons = savedComparisons + s*s;
                continue;
            end
            for k = 1:s
                tempSol(3,1:4) = [0,0,0,0]; //(this is probably redundant)
                tempSol(4,1:4) = [0,0,0,0]; //Reset
                //if modulo(k,100) == 0 then
                //    disp(k,"k=")
                //end
                if solutions(k,1,3) == 0 then
                    continue;
                else
                    tempSol(3,1:4) = solutions(k,1:4,3)
                end
                n = size(unique(abs(tempSol))); n = n(1);
                //n = number of unique solutions
                if (n > 8) then
                    //Obviously, if we already have 8 different indices,
                    //we can't add more and end up with fewer than 8
                    savedComparisons = savedComparisons + s
                    continue;
                end
                for h = 1:s
                    tempSol(4,1:4) = [0,0,0,0]; //this is probably redundant
                    if solutions(h,1,4) == 0 then
                        continue;
                    else
                        tempSol(4,1:4) = solutions(h,1:4,4)
                    end
                    completedComparisons = completedComparisons + 1;
                    n = size(unique(abs(tempSol))); n = n(1);
                    //n = number of unique solutions
```

```
                    if (n > 8) | (numberOfZeros(tempSol) == 0 & n > 7) then
                        //We want less than seven solutions, but the zeros in
                        //the array are counted so really less than eight
                        continue;
                    end
                    // deal with good algorithms
                    alg(1:4,1:4,algCounter) = tempSol; //"alg" holds the algorithms
                    algCounter = algCounter + 1;
                    disp("Algorithm found!")
                    disp(tempSol)
                    disp("i,j,k,h =")
                    disp(h,k,j,i)
                    disp("##########")
                end
            end
        end
    end
    disp("Saved comparisons and completed comparisons:")
    disp("#######", savedComparisons, "########")
    disp("#######", completedComparisons, "########")
    algorithms = alg;
endfunction

//de/indexify2 has "improved" interaction with negatives (can ignore certain aspects)
function[index] = indexify2(solutionRectangle)
    negatives = [0, 0; 0, 0];
    negativeIndex = 0;
    //Negatives on these quadrants are always arbitrary and can be ignored
    solutionRectangle(1,2) = abs(solutionRectangle(1,2));
    solutionRectangle(2,1) = abs(solutionRectangle(2,1));
    for i = 1:2
        if abs(solutionRectangle(i,i)) >= 21 | modulo(solutionRectangle(i,i),5) == 0
            solutionRectangle(i,i) = abs(solutionRectangle(i,i))*sign(solutionRectangle(3-i,3-i));
            //For zero-row/col entries, the sign does not matter,
            //so we set it equal to the other sign to minimize the
            //index and make the translation consistent
        end
    end
    negatives = sign(solutionRectangle);
    select negatives
    case [1, 1; 1, 1] then
        negativeIndex = 1;
    case [-1, 1; 1, -1] then
        negativeIndex = -1;
    case [1, 1; 1, -1] then
        negativeIndex = 2;
    case [-1, 1; 1, 1] then
        negativeIndex = -2;
    else
        disp("Error determining negatives in indexify2")
        pause;
    end;
    index = sign(negativeIndex)*(abs((solutionRectangle(1,1))*25) + ..
    abs(solutionRectangle(2,2)) + (abs(negativeIndex))*(651));
endfunction

function[solutionRectangle] = deindexify2(index)
    negativeIndex = fix(index/651); //floor does not interact as we want with negative numbers
    select negativeIndex
    case 1 then
        negatives = [1, 1; 1, 1];
    case 2 then

        negatives = [1, 1; 1, -1];
    case -1 then
        negatives = [-1, 1; 1, -1];
    case -2 then
        negatives = [-1, 1; 1, 1];
    case 0 then
        negatives = [0,0;0,0];
    else //invalid index
        negatives = 0;
    end;
    index = abs(modulo(index,651)); //remove the negative component of the index
    //We need 1 and 4 to calculate indices 2 and 3, we do them first
    ind1 = (fix((index-1)/25));
    ind4 = (modulo((index-1),25)+1);
    ind2 = (modulo((ind4-1),5)+1 + (fix((ind1-1)/5))*5);
    ind3 = (modulo((ind1-1),5)+1 + (fix((ind4-1)/5))*5);
    if modulo(ind1,5) == 0 | abs(ind1) > 20 then
        negatives(1,1) = 1;
    end
    if modulo(ind4,5) == 0 | abs(ind4) > 20 then
        negatives(2,2) = 1;
    end
    solutionRectangle = [ind1, ind2; ind3 ,ind4].*negatives;
    //elementwise multiplication to "apply" proper negatives
endfunction

function[reindexed] = reindexify(originalIndex)
    num = prod(size(originalIndex));
    reindexed = zeros(originalIndex);
```

```
    for i = 1:num
        if num > 10000 & modulo(i,2000) == 0 then
            disp(num,"of",i) //progress report
        end
        if originalIndex(i) <> 0 then
            reindexed(i) = indexify2(deindexify(originalIndex(i)));
        end
    end
endfunction

//Takes an N x 4 x 4 hypermatrix of processed indices--sorts it and removes duplicate rows
function[sorted] = sortAndTrim(processed)
    sortpro = gsort(processed,"lr","d");
    s = size(sortpro); s = s(1);
    for j = 1:4
        for i = 2:s
            if sortpro(i,1:4,j) == sortpro(i-1,1:4,j) then
                sortpro(i-1,1:4,j) = [0,0,0,0];
            end
        end
    end
    for j = 1:4
        k = 1;
        disp(j,"j=")
        for i = 1:s
            if sortpro(i,1,j) <> 0 then
                sorted(k,1:4,j) = sortpro(i,1:4,j);
                //removes all the scattered zero rows (wherever they may be)
                k = k + 1;
            end
        end
    end
endfunction

//Main
stacksize(100000000)
solutions = 0;
timer();
for i = 1:4
    all = allGridsForE(i);
    //timer()
    s = size(all); s = s(3);
    for j = 1:s
        if (modulo(j,25) == 0) then //change this to change progess update frequency
            disp("###########",j,"###########") //Progress counter
            //timer()
        end
        temp = findAllGrids(all(:,:,j));
        t = size(temp);
        solutions(1:t(1),1:t(2),j,i) = temp;
    end
end
findGridsTime = timer()
processed = processSolutions(solutions);
disp("time for processSolutions:")
processSolutionsTime = timer()
reprocessed = reindexify(processed);
reinTime = timer()
trimmed = sortAndTrim(reprocessed);
sortTrimTime = timer()
myAlgs = matchSolutions2(trimmed)
disp("Time for matchSolutions2:")
matchSolutionsTime = timer()
```