On the Use of a Machine Learning (ML) Model to Improve the Time Complexity of Gaussian Move Extraction on a Square Matrix for Inversion

Anay Aggarwal

Stoller MS

Abstract

In Linear Algebra, it has been understood that Gaussian Elimination is by far the fastest way to manually invert a matrix. However, there is no efficient implementation for this process in software, because it is not systematic – it requires intuition. The applications of matrix inversion (in cryptography, computer graphics, and machine learning) require it to be implemented in software, so this is a problem. In this project, an efficient implementation of Gauss Jordan Elimination in software was created to address this deficiency. Gaussian Elimination defines a set of "moves" that can be performed, and the more optimized the sequence of moves is, the faster it is to invert the matrix. The current software implementations of Gaussian Elimination require approximately n^3 "moves", as opposed to the approximate n^2 moves by hand. The current fastest non-Gaussian Elimination method for matrix inversion runs at around $O(n^{2.3})$. The first step was to implement matrices in code. Then, the Minimax algorithm was used to compute the fastest Gaussian Elimination process to find the inverse of each matrix in a data set. A pattern recognition algorithm was then applied to determine if the results followed an explicit formula. It turns out that it did, in fact, follow an explicit formula in the form of a piecewise function. This formula ran exactly n^2 moves, a great improvement from the previous n^3.

Background

There are multiple ways to calculate the inverse of a square matrix. The first is simply guessing and checking matrices, which is very inefficient. The second involves cofactors, adjugates, and determinants. This is also quite inefficient, but it can be done systemically. By hand, the method that is regarded as the most efficient is the Gauss-Jordan elimination method. However, the Gauss-Jordan method is difficult to implement in code because it requires intuition. Luckily, we have Machine Learning for that. Inverses of matrices can be used to solve systems of equations. Matrix inversion also plays an important role in Computer Graphics and cryptography.

Guessing and Checking, while being a valid way to invert a matrix, is extremely inefficient. In fact, it is impossible to measure it's time complexity without using other methods. You may ask how to do the "checking" part. The inverse of a matrix $\mathbb{A}$ is defined as the matrix $\mathbb{A}^{-1}$ such that the following holds:

$$\mathbb{A} \cdot \mathbb{A}^{-1} = I,$$

where I is the identity matrix (Weisstein & Stover, n.d.). The $n \times n$ identity matrix is defined as the following:

$$I_{i,j} = 0 \text{ if } i \neq j$$

$$I_{i,j} = 1 \text{ if } i = j$$

For example, $I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

Let's attempt to find the time complexity given a set S such that $\mathbb{A}^{-1}_{i,j} \in S \forall \mathbb{A}^{-1}_{i,j} \in \mathbb{A}^{-1}$. Without Loss of Generality, suppose that $\mathbb{A}$ is an $n \times n$ matrix. Notice that we will loop through each of the elements in $\mathbb{A}^{-1}$ (n^2 elements) $|S|$ times ($|S|$ denotes the cardinality, number of elements, of the set S). Thus the complexity for the guessing is $O(n^2|S|)$. To check, we must multiply the two matrices. As of October 2020, the matrix multiplication algorithm with the best time complexity runs in $O(n^{2.3728596})$ time (multiplying two $n \times n$ matrices), according to Alman & Williams, 2020. Multiplying the complexities, we get a final runtime of $O(n^{4.3728596}|S|)$. Notice that often, $S \not\subseteq \mathbb{Z}^+$ or even $S \not\subseteq \mathbb{Z}$ altogether, so this is not very feasible.

The second method to invert a matrix involves a few new concepts. It utilizes the fact that

$$\mathbb{A}^{-1} = \frac{1}{\det(\mathbb{A})} \operatorname{adj}(\mathbb{A}),$$

where $\det(\mathbb{A})$ and $\operatorname{adj}(\mathbb{A})$ are the determinant and adjugate of $\mathbb{A}$, respectively. To find the determinant, we use minors. For each $\mathbb{A}_{i,j}$, we can systematically pop out all $\mathbb{A}_{i,k}$ and $\mathbb{A}_{j,k}$ for $1 \leq k \leq n$ ($\mathbb{A}$ is an $n \times n$ matrix). We then calculate the determinant of the remaining matrix, and assign that to $\mathbb{A}_{i,j}$, according to Math Is Fun, n.d. In more formal terms,

$$\mathbb{B}_{i,j} = \det(\mathbb{A}_{x,y}), x \neq i, y \neq j,$$

where $\mathbb{B}$ is now the matrix of minors. To convert this to the matrix of cofactors ($\mathbb{C}$), for each i, j with $i + j = 0 \pmod 2$, we assign $\mathbb{C}_{i,j}$ to $\mathbb{B}_{i,j}$, and for each i, j with $i + j = 1 \pmod 2$, we assign $\mathbb{C}_{i,j}$ to $-\mathbb{B}_{i,j}$ ($i, j \in \mathbb{Z}$, of course).

Alternatively, we could simply multiply each element in $\mathbb{B}_{i,j}$ by $(-1)^{i+j}$ to get $\mathbb{C}$, according to Rusczyk & Lehoczky, 2017. "Now 'Transpose' all elements of the previous matrix... in other words swap their positions over the diagonal (the diagonal stays the same)" (Math is Fun, n.d.). The result is the adjugate matrix. Now choose $q \in \{1,2,3,...,n\}$ (it doesn't matter which q, you choose, the result will be the same). We have

$$\det(\mathbb{A}) = \sum_{r=1}^{n} (\mathrm{adj}(\mathbb{A})_{q,r} \cdot \mathbb{A}_{q,r}).$$

The determinant of $\mathbb{A}$ can also be written as $\underline{\mathbb{A}}$, as per Rusczyk & Lehoczky, 2017. It is important to note that $\mathbb{A}^{-1}$ does not exist if $\underline{\mathbb{A}} = 0$. We can see that the runtime in this method is at least bounded, be it still large.

The method that we will be focusing on in this project is commonly known as Gauss-Jordan elimination. It is the most common way to invert a matrix by hand. "To apply Gauss-Jordan elimination, operate on a matrix

$$[\mathbb{A}\ \mathbb{I}] = \begin{bmatrix} a_{11} & ... & a_{1n} \\ a_{21} & ... & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & ... & a_{nn} \end{bmatrix} \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 1 \end{bmatrix},$$

where $\mathbb{I}$ is the identity matrix, and use gaussian elimination to obtain a matrix of the form

$$\begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 1 \end{bmatrix} \begin{bmatrix} b_{11} & ... & b_{1n} \\ b_{21} & ... & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & ... & b_{nn} \end{bmatrix}.$$

The matrix

$$\mathbb{B} = \begin{bmatrix} b_{11} & ... & b_{1n} \\ b_{21} & ... & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & ... & b_{nn} \end{bmatrix}$$

is then the matrix inverse of $\mathbb{A}$. " (Weisstein, Gauss-Jordan Elimination, n.d.). But what exactly is Gaussian elimination? It is a method of solving matrix equations of the form $\mathbf{A}x = \mathbf{b}$, according to Weisstein, Gaussian Elimination, n.d. It performs a series of row operations on a matrix. For example, given

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

we can perform an operation such as $R2 \cdot 3 + R1 \cdot 2 \rightarrow R1$. First off, notice that $R1, R2, R3$ are the first, second, and third rows of the matrix, respectively. What this does is multiply the vector $\vec{R}2 = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$, second row, by the scalar 3. It then multiplies the vector $\vec{R}1 = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ by the scalar 2. It adds the two results, and pops the result into the first row of the matrix. This is $\begin{bmatrix} 12 & 15 & 18 \end{bmatrix} + \begin{bmatrix} 2 & 4 & 6 \end{bmatrix} = \begin{bmatrix} 14 & 19 & 24 \end{bmatrix}$. Thus the new matrix is $\begin{bmatrix} 14 & 19 & 24 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$.

We must do the same to the second matrix. This yields $\begin{bmatrix} 2 & 3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. So we went from

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

to

$$\begin{bmatrix} 14 & 19 & 24 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 2 & 3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

in one "move". Our end goal is the identity matrix on the left hand side, so this "move" was pretty useless. We can do any move of the form $\mathbf{v}_1 \cdot c_1 + \mathbf{v}_2 \cdot c_2 \to \mathbf{v}_3$, for constant c_1, c_2 (need not be in $\mathbb{Z}^+$), and vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ all in matrix $\mathbb{A}$.

Building a model for this process will be difficult, since it Gauss-Jordan elimination is not systematic. First, a program that will perform row operations (e.g. $R1 \cdot 1 + R3 \cdot 4 \to R3$) will be written. Then, systematic code for Gauss-Jordan elimination will be added in order to generate a dataset of the form

$$\mathbb{A}, m_1, m_2, m_3, ..., m_k,$$

where $\mathbb{A}$ is a matrix and $m_1, m_2, ..., m_k$ are the "moves" applied to find the inverse of $\mathbb{A}$. We will include over 10,000 matrices. The native language will be C++. According to Chintamaneni, 2015, given that $\mathbb{A}$ is $n \times n$, $\max(k) = \frac{n(n+1)}{2} + \frac{2n^3+3n^2-5n}{3} = \frac{4n^3+9n^2-7n}{6}$, so this is possible. Finally, we will train a ML model with the generated dataset, and ideally, it will learn how to make the set of flawless moves to find the invert any $\mathbb{A}$. The result should have a faster runtime than the plain Gauss-Jordan model does ($O(n^3)$).

Finding the inverse of a matrix has many applications. It's biggest use is in 3D graphics, specifically rotations. According to Wikipedia, 2021, "The inverse of a rotation matrix is its transpose, which is also a rotation matrix". The matrix that rotates a point (x, y) with angle θ about the $x-$axis with respect to the origin is $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, making the altered points $(x', y') = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$. Inverting this, we get another type of rotation matrix.

Matrices are also useful in cryptography. Matrices can be used to encrypt a message, meaning that we need the inverse of the matrix to decrypt the message. The quicker we find the inverse, the shorter it will take to crack. First, we split into groups of two letters (i.e. MATH RULES $\to$ MA, TH, -R, UL, ES). Then, use $A = 1, B = 2, C = 3, D = 4, ..., Z = 26, - = 27$, to convert each pair into a 2×1 matrix (MA is $\begin{bmatrix} 13 \\ 1 \end{bmatrix}$). Finally, we multiply each of the resulting matrices by the key matrix $\mathbb{A}$. This can be anything you want, as long as it's 2×2, according to Sekhon, et.al, 2021. If $\mathbb{A}$ is $n \times n$, then we would partition MATH RULES into sets of size n. To decode a message, the steps are: "

1. Take the string of coded numbers and multiply it by the inverse of the matrix that was used to encode the message.

2. Associate the numbers with their corresponding letters." (Sekhon, et.al, 2021).

Also, according to Sekchon, et.al, 2021, "This method, known as the Hill Algorithm, was created by Lester Hill, a mathematics professor who taught at several US colleges and also was involved with military encryption. "

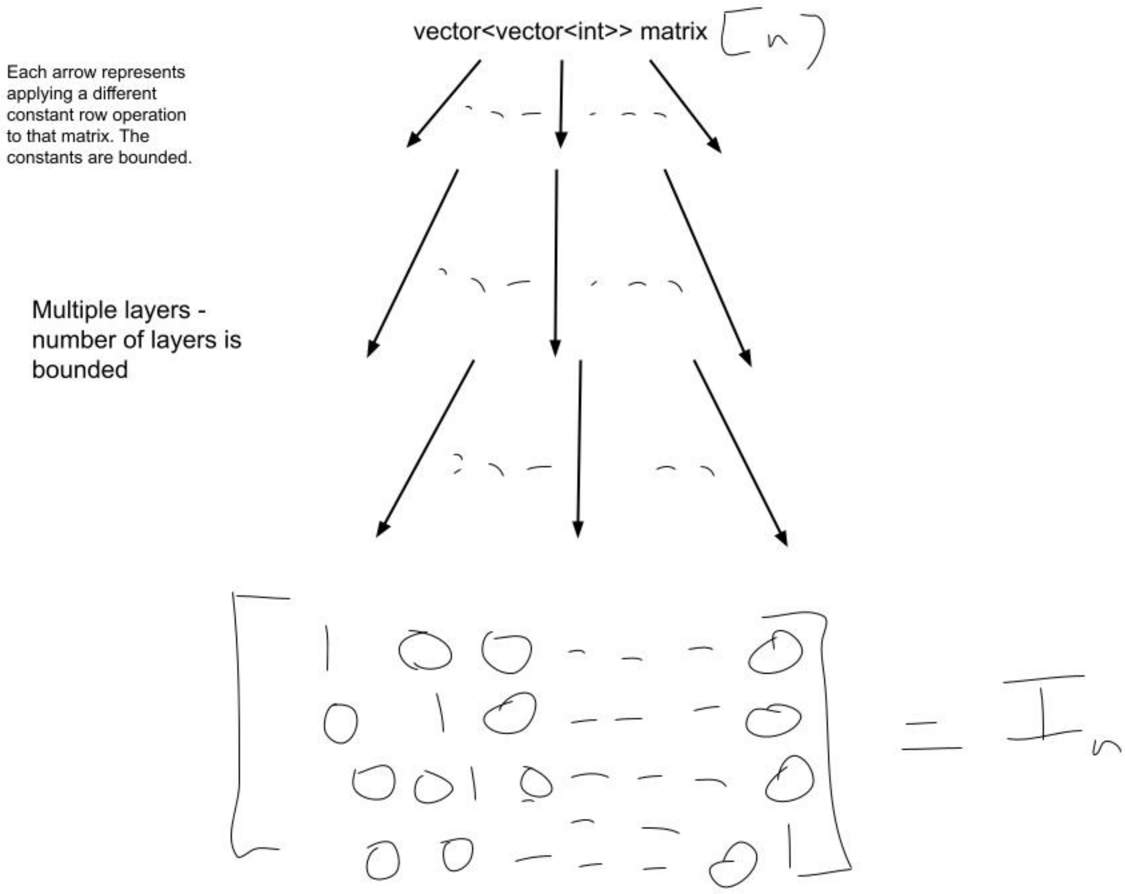
In conclusion, inverting a matrix fast is important, because it is essential to 3D graphics, and useful in the field of cryptography. We can invert a matrix in multiply ways, including guessing and checking, using the determinant-adjugate formula, and Gauss-Jordan elimination. Gauss-Jordan elimination is the most efficient, but it cannot be easily done systematically (without losing some runtime), because it requires intuition. The goal of this project is to create a machine learning model to speed up the time complexity of Gauss-Jordan elimination.

Criteria And Constraints

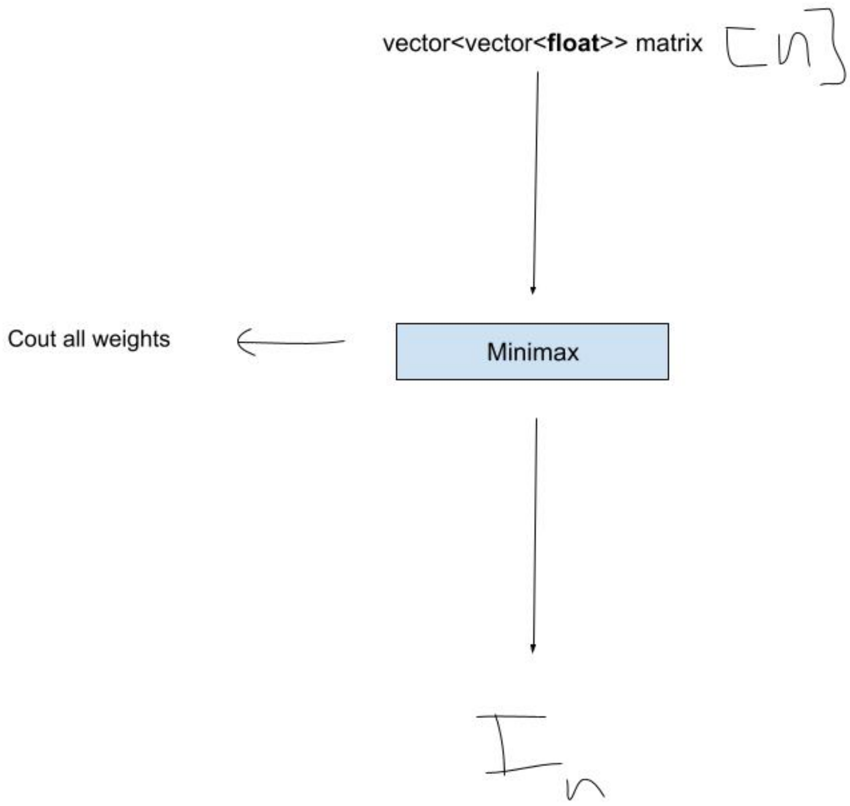
Product: An algorithm that can invert a matrix systematically. Criteria: The completed algorithm must run in less than $O(n^3)$ time (the result found by (Chintamaneni, 2015)). This means that if it runs in an asymptotic equivalent of $O(f(n))$ time, $f(N) << N^3$ for large N.

Constraints: The algorithm must run without the use of multiple processors and it should be implemented in code compatible with C++ 11.

Possible Designs



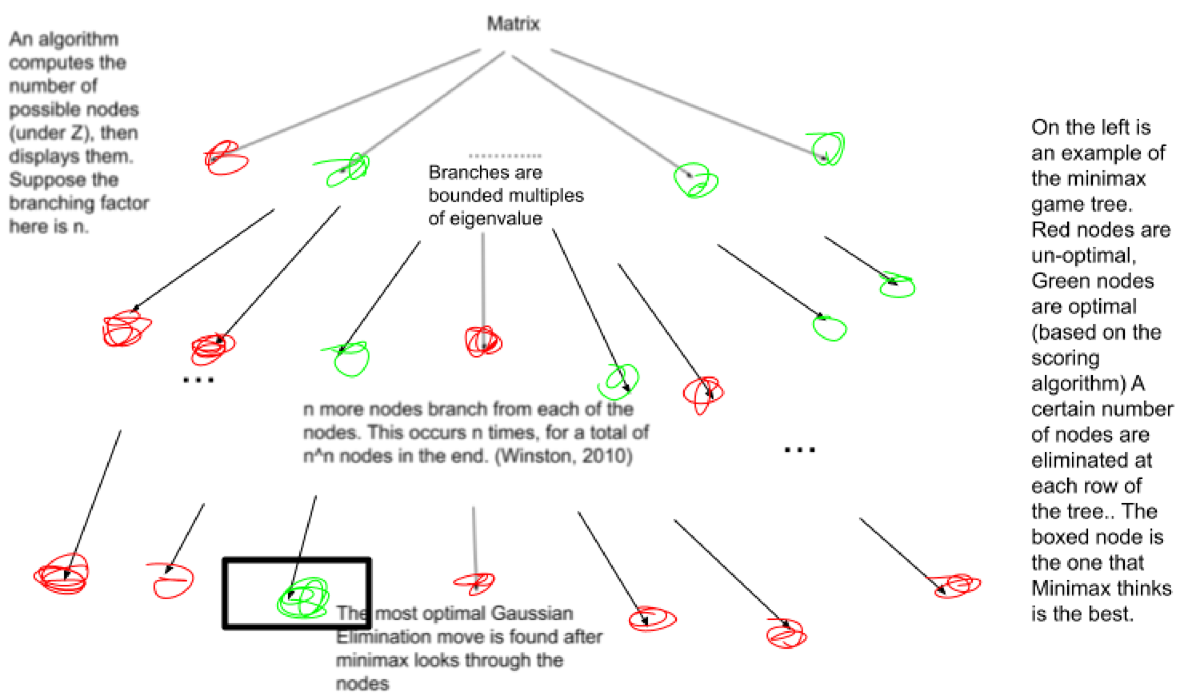
This design is intuitive and easy to implement in code. The bounds make it easy to work with. However, it's very slow and may not meet the criteria (being faster than $O(n^3)$).



This design is incredibly fast and is a novel idea. It's also tricky to implement, I will have to program the Minimax implementation myself, which will take a long time.

The 2nd design has proven to be more worthy. This is because it will easily meet my criteria, and Minimax doesn't require multiple processors, so one of the constraints can be dealt with. The other one is harder, but I figured out a way to use Minimax without upgrading to a later version of C++ (although this method is very hard to program in). For the other design, it will take a long time to process the algorithm itself, and the finished product may not even meet the criteria, which is why I chose the second design.

Minimax is a widely-used ML model in Game Theory. It is specifically prominent in Chess AI's (Winston, 2010). It is a tree-search algorithm. The essential idea is that it can look through a few nodes of a tree and decide to only search the most promising branches sprouting from that node. These branches are chosen using a scoring algorithm (which can be found in my code under the "scoringalg" function), a linear scoring polynomial on the nodes. The major innovation here is that I was able to represent inverting a matrix as a game with a game tree. Below is said tree.



Procedure

1. Implement matrices in C++
2. Implement matrix operations in C++
3. Loop through all possible combinations to generate a dataset to train the ML model on. Note: this can only be done for a few matrices.
4. Implement the minimax algorithm
5. Train the algorithm on the dataset
6. Use the minimax algorithm to generate a much larger dataset (10,000 matrices for each of 2 x 2, 3 x 3, . . . , 10 x 10 matrices)
7. Use a pattern-detection algorithm to find patterns in the dataset given the matrices
8. Write a mathematical conjecture using the pattern
9. Prove the conjecture
10. Implement the conjecture in code

Detailed information on specific aspects of the procedure can be found here:

https://github.com/Mathemagician123/RowReductionProject

Result

Completed algorithm:

```cpp
#include <math.h>

#include <algorithm>
#include <array>
#include <cassert>
#include <fstream>
#include <functional>
#include <iostream>
#include <vector>

/*
 INSTRUCTIONS
 ------------
 Construct a Matrix A via a vector of a vector. Run the "generatelooserdataset"
 algorithm that will run the algorithmic pattern found by minimax. This will
 compute the row operations required to invert the matrix. You can manually
 throw these onto the matrix.
 */

using Matrix_entry = std::pair<int, int>;  // This is for 2x2 matrix
using Matrix = std::vector<Matrix_entry>;  // This is for 2x2 matrix

uint16_t counter = 9;  // For 2x2 it's 9 moves at max, with the MIT algorithm
Matrix Moves;          // Vector of moves, used in recursion for dataset

void MultiplyVectorByScalar(std::vector<int>& entry, int k) {
  for (int i = 0; i < entry.size(); i++) {
    entry[i] *= k;  // the intuitive method
  }
}

void create_identity_matrix(std::vector<std::vector<int>>& I, int n) {
  // Create n x n identity matrix
  I = std::vector<std::vector<int>>(n, std::vector<int>(n, 0));
  for (unsigned int t = 0; t < n; t++) {
    I[t][t] = 1;  // switching all diagonal elements to 1
  }
}
```

```

std::vector<std::vector<int>> add(std::vector<int> R1, std::vector<int> R2,
                                  std::vector<std::vector<int>> A, int index,
                                  int x, int y) {
  MultiplyVectorByScalar(R1, x);
  MultiplyVectorByScalar(R2, y);
  for (size_t i = 0; i < R1.size(); i++) {
    R1[i] += R2[i];  // Adding R1 and R2
  }
  A.erase(A.begin() + index);       // Taking away old R1
  A.insert(A.begin() + index, R1);  // Adding new R1
  return A;  // returning the matrix after a single row operation
}

bool test_if_identity(Matrix& input) {
  // Simply testing if it is {{1,0}, {0,1}} (2x2 specifically)
  assert(input.size() == 2);
  if (input[0] == std::make_pair(1, 0) && input[1] == std::make_pair(0, 1)) {
    return true;
  }
  return false;
}

int determinant(int matrix[10][10], int n) {
  int det = 0;  // simply expansion by minors
  int submatrix[10][10];
  if (n == 2)
    return ((matrix[0][0] * matrix[1][1]) - (matrix[1][0] * matrix[0][1]));
  else {
    for (int x = 0; x < n; x++) {
      int subi = 0;
      for (int i = 1; i < n; i++) {
        int subj = 0;
        for (int j = 0; j < n; j++) {
          if (j == x)
            continue;
          submatrix[subi][subj] = matrix[i][j];
          subj++;
        }
        subi++;
      }
      det = det + (pow(-1, x) * matrix[0][x] * determinant(submatrix, n - 1));
    }
  }
  return det;
```

```
}
// src:
// https://www.tutorialspoint.com/cplusplus-program-to-compute-determinant-of-a-matrix

bool isinvertible(int matrix[10][10], int n) {
  return (determinant == 0)
             ? true
             : false;  // det = 0? if yes, its invertible, otherwise, it's not
}

bool TestIfIdentity(std::vector<std::vector<int>> I) {
  // Testing if all A_(i,i)=1, and A(i,j)=0 with i=/=j for the general case
  uint32_t identity_counter = 0;
  for (int iterator = 0; iterator < I.size(); iterator++) {
    std::vector<int> Dummy = I[iterator];
    for (int iterator2 = 0; iterator2 < Dummy.size(); iterator2++) {
      if (iterator = iterator2) {
        if (Dummy[iterator2] == 1) {
          identity_counter++;
        }
      } else {
        if (Dummy[iterator2] == 0) {
          identity_counter++;
        }
      }
    }
  }
  int desired = I.size() * I.size();
  if (identity_counter == desired) {
    return true;
  } else {
    return false;
  }
}

void recursionfordataset(
    int range,
    Matrix& Dummy) {  // brute-force method to look down the game tree; this
                      // will compute a small dataset for minimax to work with -
                      // takes a while to go through
  if (counter == 0) {
    std::cout << "counter is zero\n";
    if (test_if_identity(Dummy) == true) {
      std::cout << "counter is zero, identity true. returning\n";
      return;
```

```
    }

    // if it isn't an identity matrix, clear the moves and reset the
    // counter. and, start over.
    Moves.clear();
    counter = 9;
    std::cout << "reset to 9, recursing\n";
    recursionfordataset(range, Dummy);
  }

  // 4^9*range^18/9
  // the global 'counter' is non-zero
  std::cout << "recursing for dataset\n";
  for (int x = -range; x <= range; x++) {
    for (int y = -range; y <= range; y++) {
      for (uint32_t i = 0; i < 2; i++) {
        for (uint32_t j = 0; j < 2; j++) {
          if (i != j) {
            std::cout << "i: " << i << ",j: " << j << " range: " << range
                      << "\n";
            // Dummy = add(Dummy[i], Dummy[j], Dummy, i, x, y); (removed because
            // it is bashing)
            std::cout << "before push_back\n";
            Moves.push_back(std::make_pair(x, y));
            --counter;
            std::cout << "about to recurse again\n";
            recursionfordataset(range, Dummy);
            if (x != 0 && y != 0) {
              std::cout << "x, y are non-zero, calling add again\n";
              // add(Dummy[i], Dummy[j], Dummy, i, 1 / x, 1 / y); (removed
              // because it is bashing)
            }
            std::cout << "recursing again a second time\n";
            recursionfordataset(range, Dummy);
          }
        }
      }
    }
  }
}

// x and y are the essential coefficients. The rest is looping through Dummy. We
// are using a recursion (calling the function inside itself) with a break
// condition.

```

```
void generatelooserdataset(std::vector<std::vector<int>> NewDummy) {
  // This uses my conjecture, now proven
  std::vector<std::vector<int>> I;
  std::vector<std::vector<int>> Moves2;
  create_identity_matrix(I, NewDummy.size());
  for (int row = 0; row < NewDummy.size(); row++) {
    for (int column = 0; column < NewDummy.size(); column++) {
      if (row < column) {
        if (NewDummy[row + 1][column] == 0) {
          // No dividing by 0
          NewDummy[row + 1][column] = 1;
        }
        // I = add(
        //  I[row], I[row + 1], I, row, 1,
        // float(-NewDummy[row][column]) / float(NewDummy[row + 1][column]));
        // (we don't need the inverse for our dataset!, the project simply
        // relies on the set of moves)
        Moves2.push_back(
            {row, row + 1, 1,
             float(-NewDummy[row][column]) / float(NewDummy[row + 1][column])});
      }
      if (row == column) {
        if (NewDummy[row][column] == 0) {
          // No dividing by 0
          NewDummy[row][column] = 1;
          std::cout << "In here!" << std::endl;
        };
        //         I = add(I[row], I[row + 1], I, row,
        //              float(1) / float(NewDummy[row][column]), 0);
        Moves2.push_back(
            {row, row + 1, float(1) / float(NewDummy[row][column]), 0});
      }
      if (row > column) {
        //  I = add(I[row], I[column], I, row, 1, -NewDummy[row][column]);
        Moves2.push_back({row, column, 1, -NewDummy[row][column]});
      }
    }
  }
  // Moves is filled with entries of the form {row1, row2, constant1,
  // constant2}, which we will use to train our model!
  // below just outputs everything to a bufferfile
  std::ofstream bufferfile;
  bufferfile.open("dataset.txt");
  for (auto& entry : NewDummy) {
    for (int i = 0; i < entry.size(); i++) {
```

```
      bufferfile << entry[i] << " ";
    }
    bufferfile << "\n";
  }
  bufferfile << "\n";
  for (auto& entry : Moves2) {
    for (int i = 0; i < entry.size(); i++) {
      bufferfile << entry[i] << " ";
    }
    bufferfile << "\n";
  }
  bufferfile << "\n"
             << "\n";
  bufferfile.close();
  for (auto& entry : NewDummy) {
    for (int i = 0; i < entry.size(); i++) {
      std::cout << entry[i] << " ";
    }
    std::cout << "\n";
  }
  std::cout << "\n";
  for (auto& entry : Moves2) {
    for (int i = 0; i < entry.size(); i++) {
      std::cout << entry[i] << " ";
    }
    std::cout << "\n";
  }
  std::cout << "\n"
            << "\n";
  Moves2.clear();
}

int scoringalg(std::vector<std::vector<float>> Dummy) {
  // We will score this matrix using a linear scoring polynomial
  int score = 0;
  for (int i = 0; i < Dummy.size(); i++) {
    for (int j = 0; j < Dummy.size(); j++) {
      // Compare against the identity matrix
      if (i == j) {
        // should be 1
        score += pow(Dummy[i][j] - 1, 2);
      }
      if (i != j) {
        // should be 0
        score += pow(Dummy[i][j], 2);
```

```
      }
    }
  }
  return score;
}
int counter1 = 0;
std::vector<float> scores;
std::vector<std::vector<float>> record;
std::vector<float> minimax(std::vector<std::vector<float>> Dummy, int min,
                           int max) {
  // Picture a tree. Suppose Dummy's minimum element is a, maximum is b. The set
  // of constants we can use is {a,a+1,...,b,1/a,1/(a+1),...,1/b}. So there are
  // 2*(b-a+1) branches from each node of the tree. The tree ends once n^2
  // levels have been completed (Dummy is n x n)
  // this is a naive implementation of the minimax algorithm
  std::vector<float> Constants;
  for (int i = min; i <= max; i++) {
    Constants.push_back(float(i));
    if (i != 0) {
      Constants.push_back(float(1) / float(i));
    }
  }
  Constants.push_back(float(0));
  if (counter1 == pow(Dummy.size(), 2)) {
    // Hit the end of a part on the tree
    scores.push_back(scoringalg(Dummy));
  }
  // Number of total nodes: |Constants| for each level, n^2 levels, so
  // |Constants|^(n^2), very large!
  if (counter1 == pow(Constants.size(), pow(Dummy.size(), 2))) {
    // Find minimum element, corresponding moves
    // In the record, there are n^2 moves for each element in scores
    int minimum_element = 1000;
    for (int i = 0; i < scores.size(); i++) {
      if (scores[i] < minimum_element) {
        minimum_element = scores[i];
      }
    }
    for (int i = 0; i < scores.size(); i++) {
      if (scores[i] == minimum_element) {
        return record[i];
      }
    }
  }
  for (auto& entry : Constants) {
```

```
    for (auto& entry2 : Constants) {
      for (int i = 0; i < Dummy.size(); i++) {
        for (int j = 0; j < Dummy.size(); j++) {
          Dummy = add(Dummy[i], Dummy[j], Dummy, entry, entry2);
          record.push_back({i, j, entry, entry2});
          counter1++;
          minimax(Dummy, min,
                  max);  // Recursion here, counter is breaking point
        }
      }
    }
  }
}
int main() {
  // Everything in int main() is simply testing the functions
  /* std::vector<std::vector<int>> A = {{1, 2}, {3, 4}};

   std::vector<std::vector<int>> B = add(A[0], A[1], A, 0, 1, 1);
   for (auto& entry : B) {
     std::cout << entry[0] << " " << entry[1] << std::endl;
   }

   std::vector<std::vector<int>> K;
   create_identity_matrix(K, 2);

   std::vector<std::vector<int>> Bfinal = add(K[0], K[1], K, 0, 1, 1);
   for (auto& entry : Bfinal) {
     std::cout << entry[0] << " " << entry[1] << std::endl;
   }

   bool value = TestIfIdentity(B);
   bool value2 = TestIfIdentity(Bfinal);
   std::cout << value << " " << value2 << std::endl;

   // Everything in the main section is testing the program on a random
   scenario.

   // recursionfordataset(5, Dummy); Removed because it is too bashy
   // Call new function on this
   for (int a = 1; a < 10; a++) {
     for (int b = 1; b < 10; b++) {
       for (int c = 1; c < 10; c++) {
         for (int d = 1; d < 10; d++) {
           generatelooserdataset({{a, b}, {c, d}});
         }
```

```
        }
      }
    }*/
}
```

We shall now delve into the mathematical aspect of this algorithm.

Theorem 1: The $n \times n$ matrix $\mathbb{A}$ can be row-reduced to the $n \times n$ identity matrix $\mathbb{I}$ in n^2 moves.

Definition: (*Row Reduction*). Given a matrix $\mathbb{A}$, a row reduction move consists of changing a row vector $R_i \in \mathbb{A}$ to $k_1 R_i + k_2 R_j$ for a row vector $R_j \in \mathbb{A}$ where both R_i, R_j are $1 \times n$.

Corollary 1: (*Well-known*). $\mathbb{A}\mathbb{A}_k^{-1} = e_k$ has a solution for $\mathbb{A}_k^{-1}$ which can be found by Gaussian elimination on $[\mathbb{A}|e_k]$.

Corollary 2: Row operations are independent of their individual rows.

Theorem 2: (*Gauss*). Choose a set of *row reduction "moves"* $\mathscr{S}$ for an $n \times n$ matrix $\mathbb{A}$. Then there exists a $\mathscr{S}$ such that $\mathbb{I}_{\mathscr{S}} = \mathbb{A}^{-1}$ where $\mathbb{I}$ is the $n \times n$ identity matrix.

Main Lemma: (*Gauss*). If $\mathscr{S}$ reduces $\mathbb{A}$ to $\mathbb{I}$, then that same set applied to the identity matrix produces the inverse of $\mathbb{A}$.

Proof of Lemma: By definition of matrix inverse, $\mathbb{A}\mathbb{A}^{-1} = \mathbb{I}$. Thus if $i_1, i_2, \cdots, i_n \in \mathbb{I}$ are distinct row vectors of size $1 \times n$, $\mathbb{A}\mathbb{A}_j^{-1} = i_j \forall j \in \{1, 2, 3, \cdots, n\}$. By Corollary 1, we can thus solve the original matrix equation for each row of $\mathbb{A}$. Combining this fact with Corollary 2, our lemma has been proven. $\square$

Proof of Theorem 2: Using our lemma, simply choose the set $\mathscr{S}_i$ for each $R_{i[1\times n]} \in \mathbb{A}$, and then $\mathscr{S} = \bigcup_{1 \le i \le n} \mathscr{S}_i$.

Now we must prove theorem 1.

Construction: Partition $\mathbb{A}$ into n column vectors. Define the index of each vector as the column number of that vector. Starting with the vector with the smallest index, go down the vector, then move on to the next vector. At each $\mathbb{A}_{i,k}$ perform

$$R_i \to \begin{cases} R_i - \frac{\mathbb{A}_{i,k}}{\mathbb{A}_{i+1,k}} R_{i+1} & \text{if } i < k \\ R_i \cdot \frac{1}{\mathbb{A}_{i,k}} & \text{if } i = k \\ R_i - \mathbb{A}_{i,k} R_k & \text{if } i > k \end{cases}$$

Proof: First we must prove that the result after these moves are applied on $\mathbb{A}$ is $\mathbb{I}$.

Case 1: $i < k$.

We need $\mathbb{A}_{i,k} = 0$. Notice that in the specific column,

$R_{i+1} = A_{i+1,k}$. And since $\frac{\mathbb{A}_{i,k}}{\mathbb{A}_{i+1,k}} \cdot \mathbb{A}_{i+1,k} = -\mathbb{A}_{i,k}$, we're done.

Case 2: $i = k$

Clearly, $\mathbb{A}_{i,k} \cdot \frac{1}{\mathbb{A}_{i,k}} = 1$, as desired.

Case 3: $i > k$

Notice that $\mathbb{A}_{k,k} = 1$, since we have already worked on the elements in row

k. So

$$\mathbb{A}_{i,k} - \mathbb{A}_{i,k}\mathbb{A}_{k,k} = \mathbb{A}_{i,k} - \mathbb{A}_{i,k} = 0,$$

as desired.

Now we must prove that each move preserves the other elements. By Corollary 2, this is necessary to prove theorem 1. By Induction, it suffices to prove that all $\mathbb{A}_{i,j} | j < k$ are preserved.

Case 1: $i < k$.

Notice that $\mathbb{A}_{a_1,a_2} = 0 \forall a_2 < k, a_1 \neq a_2$. Therefore, unless $i + 1 = k$, we are

subtracting $c \cdot 0 = 0$, for constant c, from each element, thus preserving

them.

So it suffices to show that this works for $i + 1 = k$. Notice that the matrix

is now

$$\begin{vmatrix} 1 & 0 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & 1 & \mathbb{A}_{i,k} & ... & 0 \\ \boxed{0} & ... & \boxed{0} & 1 & ... & 0 \end{vmatrix}.$$

Since all boxed elements are 0, we're done with this case.

Case 2: $i = k$

Notice that all previous elements are 0, so multiplying them by a constant will preserve them.

Case 2: $i > k$

Clearly, it suffices to show that $\mathbb{A}_{k,j} = 0 \forall j \in \mathbb{Z}_{<k}$. This follows immediately

from our inductive hypothesis.

Hence proven. $\square$

To see this algorithm in action, consider the arbitrary matrix $\mathbb{A} := \begin{pmatrix} 1 & 2 \\ 0 & 4 \end{pmatrix}$. First, we must consider $\mathbb{A}_{11} = 1$. Since $1 = 1$, we perform the operation $R_1 \to R_1 \cdot 1$, which doesn't alter the matrix (this is why our algorithm isn't perfect, it still has a few unnecessary moves here and there; we will address this issue in the "future goals" section). Then we address $\mathbb{A}_{12} = 2$. Since $1 < 2$, we perform $R_1 \to R_1 - \frac{2}{4}R_2 = R_1 - \frac{1}{2}R_2$. This yields the matrix $\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$. Now for $\mathbb{A}_{21}$, the move is $R_2 \to R_2 - 0R_1$, thus making it $\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$ (note that this move is redundant, yet another room for improvement). Finally, we apply the move $R_2 \to R_2 \cdot \frac{1}{4}$, making the matrix $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, which is indeed $\mathbb{I}_2$, as desired. There were 2^2 moves required, and inside each move there is one if statement combined with an arithmetic operation, which runs in $O(1)$. Thus the time complexity is $O(2^2)$. The overall moves are:

$$\begin{pmatrix} R_1 \to R_1 & R_1 \to R_1 - \frac{1}{2}R_2 & R_2 \to R_2 - 0R_1 & R_2 \to R_2 \cdot \frac{1}{4} \end{pmatrix}.$$

Throwing this onto $\mathbb{I}_2$ yields $\begin{pmatrix} 1 & -\frac{1}{2} \\ 0 & \frac{1}{4} \end{pmatrix} = \mathbb{B}$. It's not hard too confirm that $\mathbb{AB} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathbb{I}$, thus $\mathbb{B} = \mathbb{A}^{-1}$, so our algorithm correctly found the inverse of $\mathbb{A}$.

Analysis

The algorithm found by minimax can generate the Gaussian Elimination moves required to invert an n x n matrix in $O(n^2)$ time. The runtime was tested both mathematically (n^2 moves would obviously correspond to n^2 time) and by computer (giving sample matrices and recording the runtime for each). This is faster than the previous $O(n^3)$ result found by Chintamaneni, 2015. It is also faster than the current non-elimination method, which has a result of approximately $O(n^{2.3})$. The exact algorithm is on the Github page linked above. The difficulty issue that was mentioned while planning possible designs was prominent, but I was able to overcome this with sheer hard work and research. The code is quite efficient as well.

Data

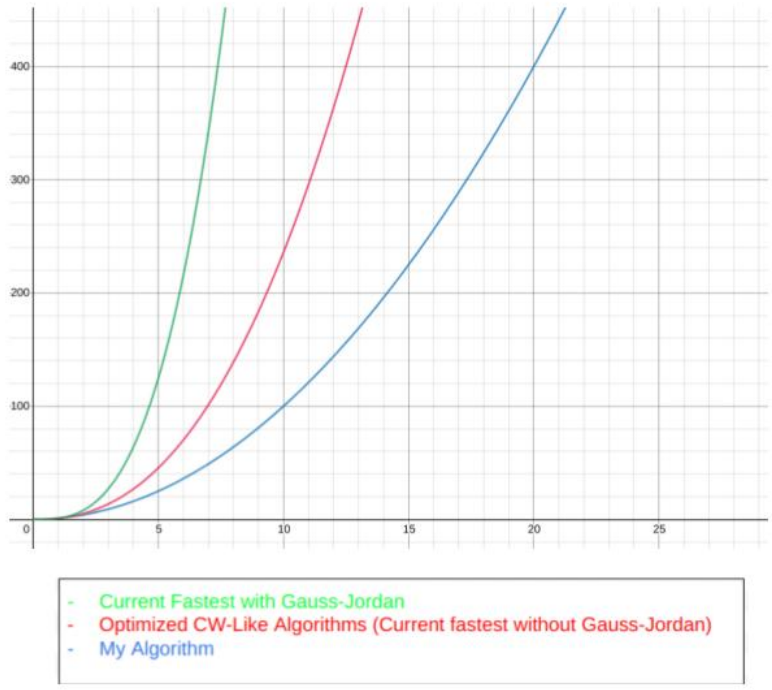
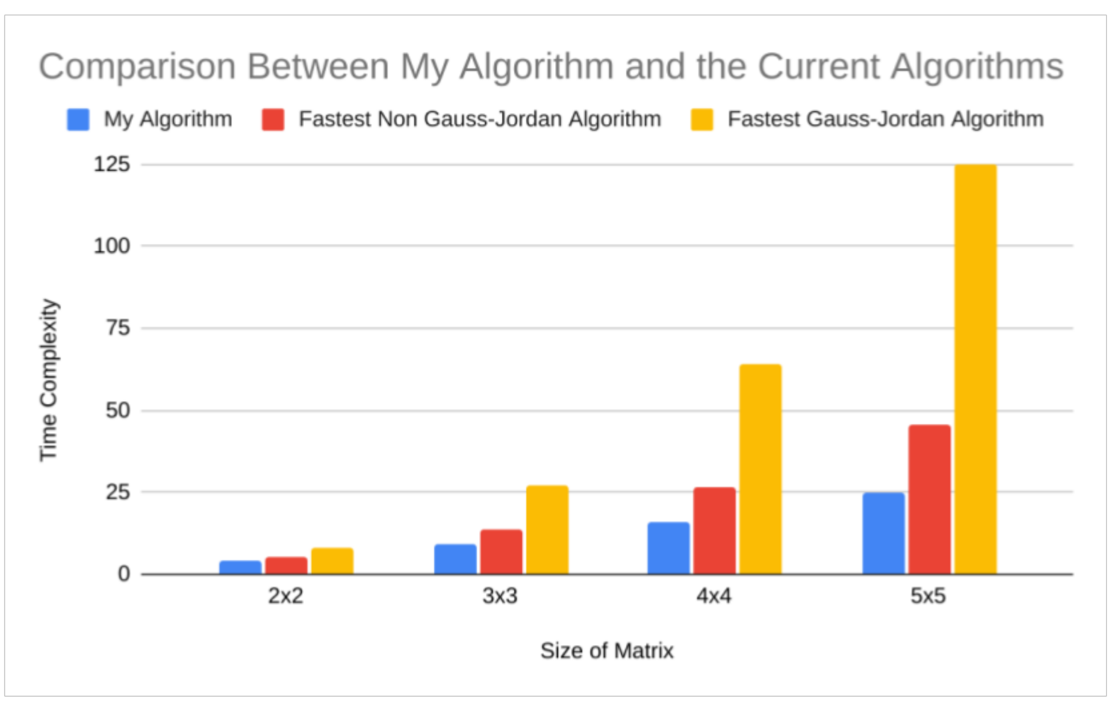


Figure 1: Above is a comparison of the runtimes of the different inversion algorithms. The y-value represents runtime and the x-value represents the size of the matrix. It is a continuous visualization of the functions.



Setbacks

1. Multiplying a vector by a scalar wasn't working correctly. I realized this after I had gone through and implemented the rest, so I had to completely restart.

2. Writing the recursion such that it wouldn't loop infinitely (i.e. creating a valid break variable + condition) was a challenge, as the recursion in the minimax algorithm was very complicated (many bugs here).

3. While I was implementing the minimax algorithm, I realized I was going to need to deal with floating point numbers because I hadn't created a division operator. I had to restructure the entire code because of this.

4. Dividing by 0. The original conjecture was vulnerable to division by zero, I had to redo it to avoid division by zero.

Future Goals

I plan to take this project to the next level by replacing the systematic algorithm that I developed in this project for an algorithm that is more reliant on artificial intelligence and leverages neural networks to improve the runtime even more. Currently, my program inputs a matrix $\mathbb{A} \in \mathbb{Q}^2$, but cannot do $\mathbb{A} \in \overline{\mathbb{Q}}^2$ or $\mathbb{A} \in (\mathbb{C} \backslash \mathbb{R})^2$. I plan on implementing a framework in C++ for matrices with irrational or complex numbers. I also aim to implement the Gaussian moves to inverse conversion as quickly as possible, so my project can attack the broader topic of matrix inversion.

References

1. Weisstein, E. W., & Stover, C. (n.d.). Matrix Inverse. Retrieved January 05, 2021, from https://mathworld.wolfram.com/MatrixInverse.html

2. Alman, J., & Williams, V. V. (2020, October 13). A Refined Laser Method and Faster Matrix Multiplication. Retrieved January 05, 2021, from https://arxiv.org/pdf/2010.05846.pdf

3. Inverse of a Matrix using Minors, Cofactors and Adjugate. (n.d.). Retrieved January 05, 2021, from https://www.mathsisfun.com/algebra/matrix-inverse-minors-cofactors-adjugate.html

4. Rusczyk, R., & Lehoczky, S. (2017). The Art of Problem Solving: Volume 2 and Beyond. Alpine, CA: AoPS.

5. Weisstein, E. W. (n.d.). Gauss-Jordan Elimination. Retrieved January 05, 2021, from https://mathworld.wolfram.com/Gauss-JordanElimination.html

6. Weisstein, E. W. (n.d.). Gaussian Elimination. Retrieved January 06, 2021, from https://mathworld.wolfram.com/GaussianElimination.html

7. Chintamaneni, K. (2015, September 23). Operations Required in Matrix Elimination. Retrieved January 06, 2021, from http://web.mit.edu/18.06/www/Fall15/Matrices.pdf

8. Rotation matrix. (2021, January 01). Retrieved January 06, 2021, from https://en.wikipedia.org/wiki/Rotation_matrix

9. Sekhon, R., & Bloom, R. (2021, January 02). 2.5: Application of Matrices in Cryptography. Retrieved January 07, 2021, from https://math.libretexts.org/Bookshelves/Applied_Mathematics/Book%3A_Applied_Finite_Mathematics_(Sekhon_and_Bloom)/02%3A_Matrices/2.05%3A_Application_of_Matrices_in_Cryptography

10. Million, E. (2007, April 12). The Hadamard Product. Retrieved from http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf

11. Winston, P. (2010, Fall). Artificial Intelligence 6.034, Massachusetts Institute of Technology. Retrieved from https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/