The Fundamentals of Utilizing Algorithms to Calculate the Determinant of a Matrix

Math 2015 (Spring 2024) Capstone

Jessica Card, Alexander Mies, Jacob Million, and Edward Yi

April 19, 2024

Abstract

In our capstone project, we will introduce a few commonly-used algorithms for calculating the determinant of a matrix by showing our Python implementations of each. First, we will provide a brief explanation on determinants as a mathematical concept. Next, we will introduce the algorithms that are commonly-used for calculating determinants, along with providing an explanation for each of these algorithms. Lastly, we will show and compare the different time complexities of each algorithm with time tests and graphs.

Four of the most commonly-used algorithms are covered in this paper: Laplace Expansion, Bareiss, Lower-upper decomposition, and Eigen-decomposition.

1 Introduction

In this paper, we introduce a few commonly-used algorithms for calculating the determinant of square matrices implemented in Python. These algorithms include Laplace Expansion, Bareiss, Lower-upper (LU) decomposition, and Eigen-decomposition.

We organized the paper as follows: Section 2 will give an overview of the mathematical concept of a determinant. Section 3 will discuss more in-depth about each of the algorithms that are commonly-used to calculate these determinants that are reviewed in Section 2. Section 4 will provide information on the Big O notation and its relation/significance in comparing time complexities for the algorithms that were presented in Section 3. Section 4 will also include information on the time complexities for the algorithms we mentioned earlier and graph results of their runtimes when we process these algorithms (in Python). To conclude, we discuss the results of our observations and analysis of these algorithms when compared to one another, while providing any lasting comments on the topic.

2 Determinants

A determinant is a mathematical object in the analysis and solution of systems of linear equations. For square matrices $\in \mathbb{R}^{nxn}$, i.e., matrices with the same number of rows and columns, the determinant indicates whether the matrix is invertible. However when the result of a determinant is equal to zero, that means the matrix is not invertible. The determinant also geometrically represents the scaling factor of the linear transformation defined by the matrix, specifically how the area (in 2D) or volume (in 3D) is scaled when vectors are transformed by the matrix.[1] The determinant of a matrix is commonly denoted as det(A), det A, or sometimes as $|A|$ such that:

$$A = \det \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

The **determinant** of a square matrix A $\in \mathbb{R}^{nxn}$ is a function that maps A onto a real number and exhibits the following properties:

1) Singular matrices are matrices with a determinant of 0.

2) Unimodular matrices are matrices with a determinant of 1.

3) The determinant of a 1x1 matrix is the element itself.

$$A = [a] \rightarrow det(A) = a$$

4) The determinant of a 2x2 matrix is defined as the product of the elements on the main diagonal minus that of the product of the elements off the main diagonal. As such:

$$\begin{vmatrix} \text{a} & \text{b} \\ \text{c} & \text{d} \end{vmatrix} = ad - bc \text{ [2]}$$

5) The determinant of the identity matrix is 1.

$$det(I) = 1$$

6) The determinant of a matrix product is the product of the corresponding determinants, which is also true for square matrices A and B of equal size.

$$det(AB) = det(A)det(B)$$

7) Determinants are invariant to transposition.

$$det(A) = det(A^T)$$

8) If A is regular (invertible), then:

$$det(A^{-1}) = \frac{1}{det(A)}$$

9) For a triangular matrix T $\in \mathbb{R}^{nxn}$, the determinant is the product of the diagonal elements, i.e.,

$$det(T) = \prod_{i=1}^{n} T_{ii}$$

10) Similar matrices possess the same determinant. Therefore, for a linear mapping $\Phi : V \rightarrow V$, all transformation matrices A_Φ of Φ have the same determinant. Thus, the determinant is invariant to the choice of basis of a linear mapping.

11) Adding a multiple of a column/row to another one does not change $det(A)$.

12) Multiplication of a column/row with $\lambda \in \mathbb{R}$ scales $det(A)$ by λ.

$$det(\lambda A) = \lambda^n det(A)$$

13) Swapping two rows/columns changes the sign of $det(A)$.

When mathematics was mainly performed by hand, the determinant calculation was considered an essential way to analyze matrix invertibility. However, contemporary approaches in machine learning use direct numerical methods that superseded the explicit calculation of the determinant. For example, we now know that inverse matrices can be computed by Gaussian elimination. Thus, Gaussian elimination can be used to compute the determinant of a matrix. Determinants are also known to play a important theoretical role, especially in regards to to eigenvalues and eigenvectors through their characteristic polynomial.[1]

3 Commonly-Used Algorithms

3.1 Laplace Expansion

Laplace Expansion, also commonly referred to as the co-factor expansion, calculates the determinant of a matrix by recursively expanding it along a row or column. In each step, it multiplies the determinant of a smaller sub-matrix (obtained by removing the current row and column) by the value of the element in the position at which the expansion is taking place. The signs of these products alternate according to the position of the element in the matrix, following a pattern that can be determined by adding the row and column indices of the element (with even sums leading to a positive sign and odd sums to a negative sign). This process continues until the determinant of 2x2 matrices can be directly computed, at which point the recursive expansion concludes.

To calculate the determinant of the matrix A,

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

We can use Laplace expansion along the first row. The determinant of A can be computed as:

$$\det(A) = a \cdot \det \begin{bmatrix} e & f \\ h & i \end{bmatrix} - b \cdot \det \begin{bmatrix} d & f \\ g & i \end{bmatrix} + c \cdot \det \begin{bmatrix} d & e \\ g & h \end{bmatrix}$$

The individual 2x2 determinants are calculated as follows:

$$\det \begin{bmatrix} e & f \\ h & i \end{bmatrix} = ei - fh \qquad \det \begin{bmatrix} d & f \\ g & i \end{bmatrix} = di - fg \qquad \det \begin{bmatrix} d & e \\ g & h \end{bmatrix} = dh - eg$$

Thus, substituting back, we get:

$$\det(A) = a(ei - fh) - b(di - fg) + c(dh - eg)$$

Below is our version of the core Laplace's algorithm in Python:

```
def calc_det(matrix):
    # We can just check the outer length because we assume the matrix is square
    # to begin with
    if len(matrix) == 2:
        # Base case, 2x2 matrix
        return ((matrix[0][0] * matrix[1][1]) - (matrix[0][1] * matrix[1][0]))

    # Result of the determinant
    sum = 0

    # Any matrix we have here is > 2x2
    for i, x in enumerate(matrix):
        # Chop out 0th column and ith row
        chopped_matrix = []
        for j, _ in enumerate(matrix):
            if i != j:
                chopped_matrix.append(matrix[j][1:])

        # We have to alternate + / - for this algorithm
        for k, _ in enumerate(x):
            if i % 2 == 0:
                if k % 2 == 0:
```

```
                sum += x[0] * calc_det(chopped_matrix)
            else:
                sum -= x[0] * calc_det(chopped_matrix)
        else:
            if k % 2 == 0:
                sum -= x[0] * calc_det(chopped_matrix)
            else:
                sum += x[0] * calc_det(chopped_matrix)

    return sum
```

The full commented code for our Laplace algorithm implementation can be viewed here.

3.2 Bareiss

The Bareiss algorithm can be thought of as a "Multi-step Integer-Preserving Gaussian Elimination" as was the paper titled by Erwin H. Bareiss, the mathematician that developed the algorithm. This algorithm can be applied to matrices with integer entries. It is unique as it uses only integer arithmetic in its procedure, and the division performed in the code below is guaranteed to result in an integer without remainder. In code, this looks three nested 'for' loops which modify the matrix in place.

The procedure is as follows: Begin at the top left-most entry. This is $M[i,i]$ for $i = 0$. Then, for $j,k = i+1$, beginning at the entry one to the right of i and one entry down, the formula: $M[j][k] = (M[j][k] * M[i][i] - M[j][i] * M[i][k])/prev$, is applied. The variable $prev$ is originally initiated to 1 and represents the previous $M[i][i]$. After iterating over all k for the second row and applying the formula, the next row is then iterated over and this repeats until the bottom right-most entry is modified. At this point, i is incremented, $M[i][i]$ is set to be the current $M[i][i]$, and the process repeats for $i = 2 : i[2][2]$, the next diagonal entry. This repeats until i reaches the second to last row and the final iteration is ran. The determinant is then found at the bottom right-most entry or $M[-1][-1]$, as the matrix is modified in place.

Like in Gaussian elimination, rows can be swapped in this procedure if necessary. If any A[i,i] becomes zero during computation, or is 0, a suitable row to swap with is searched for and swapped with if one exists. If this is done, the sign of the determinant is flipped in correspondence with each swap. Otherwise, the determinant is zero and the algorithm terminates early. The algorithm used in the code for this project functions only for an n x n matrix. Below is the core of the Bareiss algorithm, implemented in Python (row swapping and other bookkeeping code removed for simplicity):

We can use the Bareiss algorithm to compute the determinant of a matrix A:

$$A = \begin{bmatrix} 1 & -3 & 3 \\ 4 & 2 & -6 \\ 2 & 4 & 8 \end{bmatrix}$$

Determinant of A can thus be calculated as follows:

Recall the equation,

$$M[j][k] = \frac{M[j][k] \times M[i][i] - M[j][i] \times M[i][k]}{prevM[i,i]}$$

$$prevM \text{ initialized to } 1$$

$$M[i,i] = 1 \qquad M[j,k] = 2 \qquad M[i,k] = -3 \qquad M[j,i] = 4$$

$$\text{updated } M[j,k] = \frac{2 \times 1 - 4 \times (-3)}{1}$$

Updated Matrix:

$$A' = \begin{bmatrix} 1 & -3 & 3 \\ 4 & 14 & -6 \\ 2 & 4 & 8 \end{bmatrix}$$

$$\text{updated } M[j, k = k+1] = \frac{-6 \times 1 - 4 \times (3)}{1}$$

Updated Matrix:

$$A' = \begin{bmatrix} 1 & -3 & 3 \\ 4 & 14 & -18 \\ 2 & 4 & 8 \end{bmatrix}$$

At this point, j is incremented as we have reached the right-most entry of this row.

$$\text{updated } M[j = j + 1, k = i + 1] = \frac{4 \times 1 - 2 \times (-3)}{1}$$

Updated Matrix:

$$A' = \begin{bmatrix} 1 & -3 & 3 \\ 4 & 14 & -18 \\ 2 & 10 & 8 \end{bmatrix}$$

$$\text{updated } M[j, k = k + 1] = \frac{8 \times 1 - 2 \times (3)}{1}$$

Updated Matrix:

$$A' = \begin{bmatrix} 1 & -3 & 3 \\ 4 & 14 & -18 \\ 2 & 10 & 2 \end{bmatrix}$$

At this point, i is incremented so $M[i][i] = M[2][2]$. $prevM[i, i]$ remains 1.

$$\text{updated } M[j = i + 1, k = i + 1] = \frac{2 \times 14 - 10 \times (-18)}{1}$$

Updated Matrix:

$$A' = \begin{bmatrix} 1 & -3 & 3 \\ 4 & 14 & -18 \\ 2 & 10 & 208 \end{bmatrix}$$

At this point, there is nothing further we can apply the formula to.

The determinant can be found at $M[-1][-1]$ so the solution is 208.

Below is our version of the core Bareiss algorithm in Python:

```
# Simplified Bareiss Algorithm:

for i in range(N-1): # where N is the size of the matrix

    # ensure non-zero M[i][i]
    # flip sign if zero M[i][i] and suitable swap row exists, swap
    # return 0 if zero M[i][i] and suitable swap row does not exist

    for j in range(i+1, N):
        for k in range(i+1, N):

            # applying the formula
            M[j][k] = (M[j][k] * M[i][i] - M[j][i] * M[i][k]) // prev

    prev = M[i][i]  # updating for next iteration

# returning the determinant, located at the bottom rightmost position
return sign * M[-1][-1]
```

The full commented code for our Bareiss algorithm implementation can be viewed here.

3.3 Lower-upper decomposition

LU decomposition is an algorithm that can be used for solving systems of linear equations, the inversion of matrices, and in our case, computing the determinant of a square matrix. Our goal, given a matrix A, is to decompose A into three individual components: a lower triangular matrix L, an upper triangular matrix U (not unlike an upper triangular matrix we form with Gaussian elimination), and a permutation matrix P that tracks row swaps – such that $P * A = L * U$.

In our particular implementation, we initialize square matrix A and a permutation vector P with indices $0 - N$. The last element of P, $(P[N])$, counts the number of row swaps to determine the sign of the determinant. For picking the pivots, we iterate through each column i and search for the element with the maximum absolute value in that column. In swapping rows, if the row with the maximum element $imax$ is not the current row i, we swap that row with $imax$ – not only swapping within A but modifying the permutation vector P (important for determining sign at the end).

After we pivot, we look at all rows below i and each element below the specified pivot in the current column is divided by the pivot, forming part of the lower triangular (L) matrix. We then update the remaining elements for each row by subtracting a multiple in the current row (close to Gaussian elimination) to effectively construct both the upper and lower triangular matrices in place. Since the determinant of a triangular matrix is the product of it's diagonal elements, and U is the upper triangular after the decomposition, the determinant can be easily calculated once we've decomposed A into LU and determine the sign with P.

In LU-Decomposition, our goal is to decompose a matrix A into an upper matrix U and a lower matrix L:

$$A = \begin{bmatrix} A00 & A01 & A02 \\ A10 & A11 & A12 \\ A20 & A21 & A22 \end{bmatrix} = L \begin{bmatrix} 1 & 0 & 0 \\ L10 & 1 & 0 \\ L20 & L21 & 1 \end{bmatrix} \cdot U \begin{bmatrix} U00 & U01 & U02 \\ 0 & U11 & U12 \\ 0 & 0 & U22 \end{bmatrix}$$

After decomposing, the the determinant of A can easily be calculated by taking the product on the diagonals of L and U:

$$= L \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \cdot U \begin{bmatrix} U00 & & \\ & U11 & \\ & & U22 \end{bmatrix}$$

Where $det(A) = \det(L) \cdot \ \det(U) \ = 1 \cdot 1 \cdot 1 \cdot U00 \cdot U11 \cdot U22$

As an example, lets define matrix A as:

$$\begin{bmatrix} 3 & 2 & 4 \\ 2 & 0 & 2 \\ 4 & 2 & 3 \end{bmatrix}$$

We first obtain the upper-triangular matrix U by applying Gaussian elimination to A as well as bookkeeping the values we're multiplying by to be stored in L:

$$R_2 \leftarrow R_2 - \frac{2}{3} R_1 \quad [\mathrm{L}_{2,1} = \frac{2}{3}]$$

$$\begin{bmatrix} 3 & 2 & 4 \\ 0 & \frac{-4}{3} & \frac{-2}{3} \\ 4 & 2 & 3 \end{bmatrix}$$

$$R_3 \leftarrow R_3 - \frac{4}{3} R_1 \quad [\mathrm{L}_{3,1} = \frac{4}{3}]$$

$$\begin{bmatrix} 3 & 2 & 4 \\ 0 & -4/3 & -2/3 \\ 0 & -2/3 & -7/3 \end{bmatrix}$$

$$R_3 \leftarrow R_3 - \frac{1}{2}R_2 \quad [\mathrm{L}_{3,2} = \frac{1}{2}]$$

$$\begin{bmatrix} 3 & 2 & 4 \\ 0 & -4/3 & -2/3 \\ 0 & 0 & -2 \end{bmatrix}$$

As a result, we've formed both the upper-triangular and lower-triangular matrices derived from A:

$$U = \begin{bmatrix} 3 & 2 & 4 \\ 0 & -4/3 & -2/3 \\ 0 & 0 & -2 \end{bmatrix}, L = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 4/3 & 1/2 & 1 \end{bmatrix}$$

With this, we can calculate the determinant by taking the product of the diagonals in L,U: $det(A) = det(L) \cdot \det(U) \ = 1 \cdot 1 \cdot 1 \cdot 3 \cdot (-4/3) \cdot -2 = 8$

Below is our version of the core LU decomposition algorithm in Python:

```
def LUPDecompose(A, Tol=1.0e-9):
    N = len(A)
    P = np.arange(N+1)
    for i in range(N):
        maxA = 0.0
        imax = i
        for k in range(i, N):
            absA = abs(A[k][i])
            if absA > maxA:
                maxA = absA
                imax = k
        if maxA < Tol:
            raise Exception("Matrix is degenerate")
        if imax != i:
            P[i], P[imax] = P[imax], P[i]
            A[[i, imax]] = A[[imax, i]]  # swap rows
            P[N] += 1
        for j in range(i+1, N):
            A[j][i] /= A[i][i]
            A[j][i+1:N] -= A[j][i] * A[i][i+1:N]
    return P
```

The full commented code for our LU decomposition algorithm implementation can be viewed here.

3.4 Eigen-decomposition

Provided that a matrix is diagonalizable, we can also use a method called Eigen-decomposition to compute the determinant. In this method, we aim to decompose a square matrix A into a product of three matrices: $A = PDP^-1$ where P is a matrix made up of the eigenvectors of A, and D is a diagonal matrix whose elements are all the eigenvalues of A. The determinant then is the product of its eigenvalues. More simply put, we:

1) Compute all eigenvalues of A, which in turn become the diagonal elements of D.

2) Calculate the product of the eigenvalues.

Because the determinant of a matrix is equal to the product of its eigenvalues, Eigen-decomposition can be incredibly efficient on smaller matrices, with the exception that the provided matrix is already diagonalizable.

To calculate the determinant of a matrix using it's eigenvalues, we must first calculate the eigenvalues. Let A be the square matrix:

$$A = \begin{bmatrix} 2 & 1 & -2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

We start with defining the characteristic equation for our square matrix A:

$$det(A) = (A - \lambda I)$$

Where λ is a scalar and I is an identity matrix of an appropriate size. Plugging this into our characteristic equation yields:

$$\begin{bmatrix} 2 & 1 & -2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2-\lambda & 1 & -2 \\ 1 & -\lambda & 0 \\ 0 & 1 & -\lambda \end{bmatrix}$$

When we subtract λI from A, we essentially scale the identity matrix by λ and subtract the result from A. Our next step is to then calculate the determinant of the matrix we created using the characteristic equation. This can be represented with:

$$det(A - \lambda I) = (2-\lambda)[(-\lambda)(-\lambda) - (0)(1)] - 1[-\lambda - 0] + (-2)[1 - 0]$$

$$= (2-\lambda)[\lambda^2] - 2 + \lambda$$

Setting this to 0 allows us to solve for λ:

$$0 = (2-\lambda)[\lambda^2] - (-\lambda + 2)$$

$$0 = (2-\lambda)[\lambda^2 - 1]$$

$$0 = (2-\lambda)(\lambda + 1)(\lambda - 1)$$

From here, we can determine the three eigenvalues from our original matrix A by plugging in the 3 values of λ that satisfy the equation:

$$\lambda_1 = 2, \lambda_2 = -1, \lambda_3 = 1$$

Calculating the product of these three eigenvalues yields the determinant of matrix A:

$$2 \cdot -1 \cdot 1 = -2$$

Below is our version of the core Eigen-decomposition algorithm in Python:

```
eigenvalues, _ = np.linalg.eig(A)
determinant = np.prod(eigenvalues)
```

The full commented code for our Eigen-decomposition algorithm implementation can be viewed here.

4 Runtime and Time Complexity Comparisons

Big-O, denoted as $O(f(n))$, where $f(n)$ is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size n, is used to describe the performance or complexity of an algorithm. Specifically, it is used to describe the **worst-case scenario** in terms of time or space complexity. Big-O notation provides a way to compare the performance of difference algorithms and data structures, and to predict how they will behave as the input size increases. Several reasons that Big-O notation is important include:

1) helping to analyze the efficiency of algorithms

2) providing a way to describe the **runtime** or **space requirements** of an algorithm grow as the input size increases

3) allowing programmers to compare different algorithms and choose the most efficient one for a specific problem

4) helping in understanding the scalability of algorithms and predicting how they will perform as the input size grows

5) enabling developers to optimize code and improve overall performance [4]

As we can see in Figure 4, $O(n!)$ represents the **worst-case** scenario when the algorithm is least efficient and $O(1)$ represents the **best-case** scenario when the algorithm is **most** efficient.

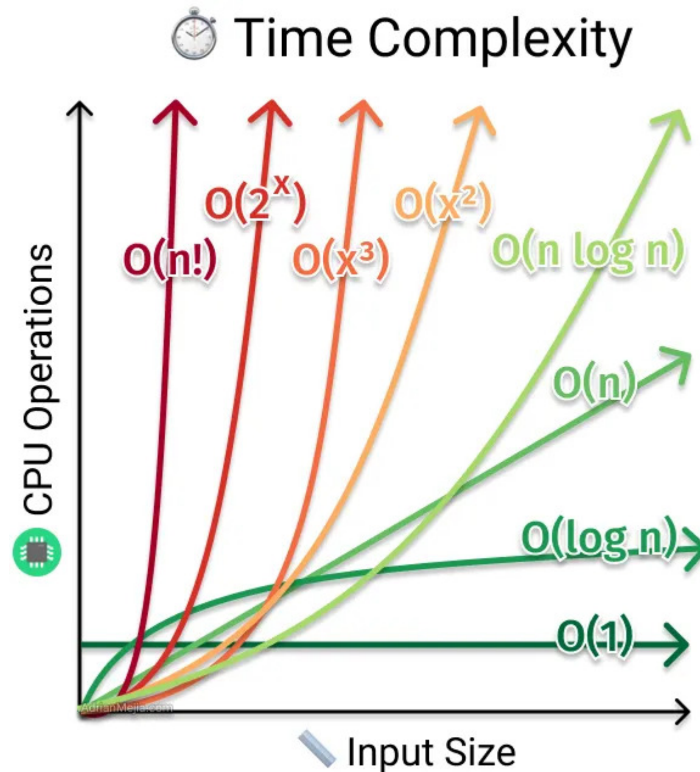

Figure 1: Big-O Time Complexity Comparison Chart [3]

4.1 Laplace Expansion

Time complexity: $O(n!)$

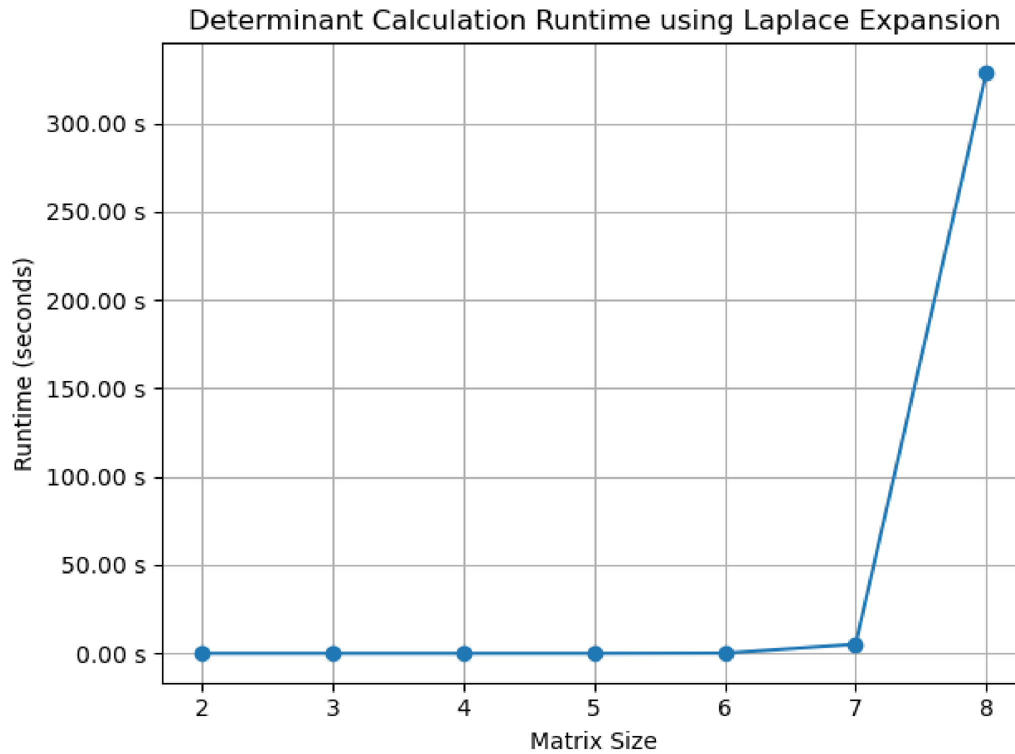

Figure 2: Determinant Calculation Runtime using Laplace Expansion

This graph shows the factorial nature of Laplace. Unlike the other algorithms noted in this paper whose test matrices went over size 100x100, we could only run test cases on Laplace expansion up to matrices of size 8x8. This is due to the factorial nature of Laplace - 7x7 took than 10 seconds, whereas 8x8 jumped to over 5 minutes. If we attempted a 9x9 sized matrix, it would not finish calculating in any sort of reasonable amount of time.

Laplace expansion has a time complexity of $O(n!)$ (factorial time complexity) due to the nature of its recursive expansion. This complexity arises because, at each level of recursion, the number of determinant calculations grows factorially. Specifically, you start with n choices, then for each of those, you have $n-1$ choices, and so on. This sequence of choices produces a factorial number of operations, as illustrated by the product $n*(n-1)*(n-2)*\ldots*2*1$, which is $n!$.

This recursive process is quite inefficient for large n due to its exponential growth in the number of operations required, leading to the factorial time complexity. However, this method of calculating the determinant is easy for the human brain to understand, so we often learn to factor by hand using this form of calculation.

4.2 Bareiss Algorithm

Time complexity: $O(n^3)$

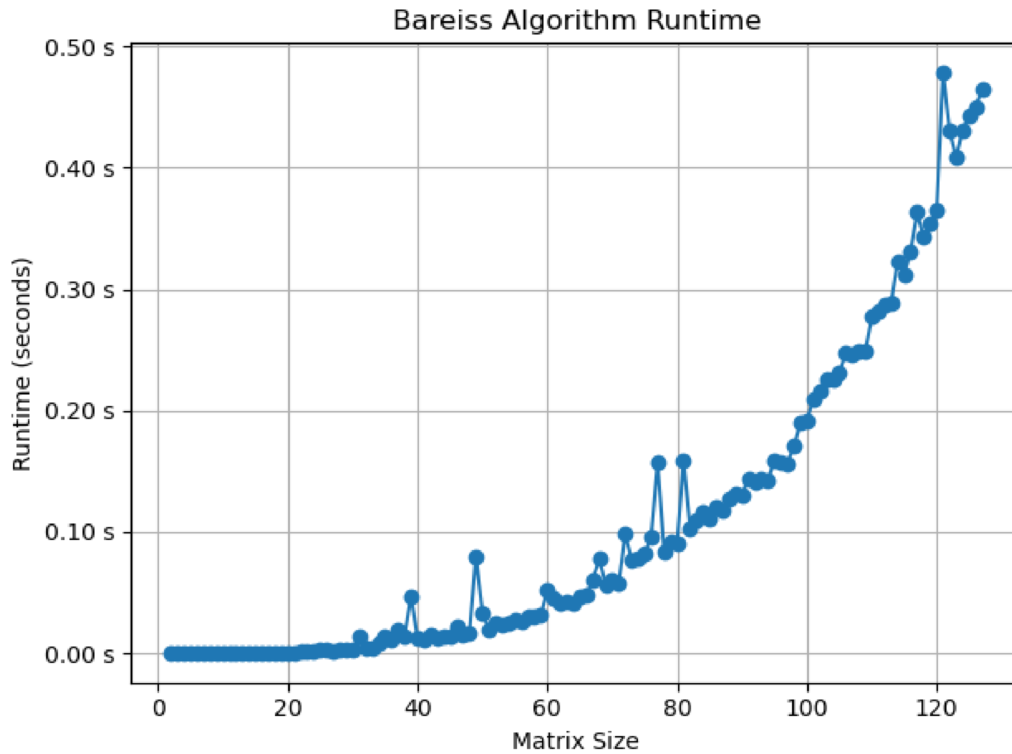

Figure 3: Determinant Calculation Runtime using the Bareiss algorithm

The graph shows the polynomial nature of the Bareiss algorithm. This algorithm requires $O(n^3)$ time complexity, similar to Gaussian Elimination, and is significantly faster than Laplace Expansion, which requires O(n!) time.

The runtime the Bareiss algorithm requires can be thought of quite simply by the use of 3 nested 'for' loops. Iterating over each $M[i,i]$ entries requires $O(n)$ time, iterating over all elements of a row requires $O(n)$ time, and iterating over all columns requires $O(n)$ time. The argument could be made that for every iteration of i, the number of computations decreases. This is true, but the number of computations is still on the order of the input size. As these loops are all nested, their product is taken to arrive at the time complexity for Bareiss: $O(n) * O(n) * O(n) = O(n^3)$.

4.3 LU decomposition

Time complexity: $O(n^3)$

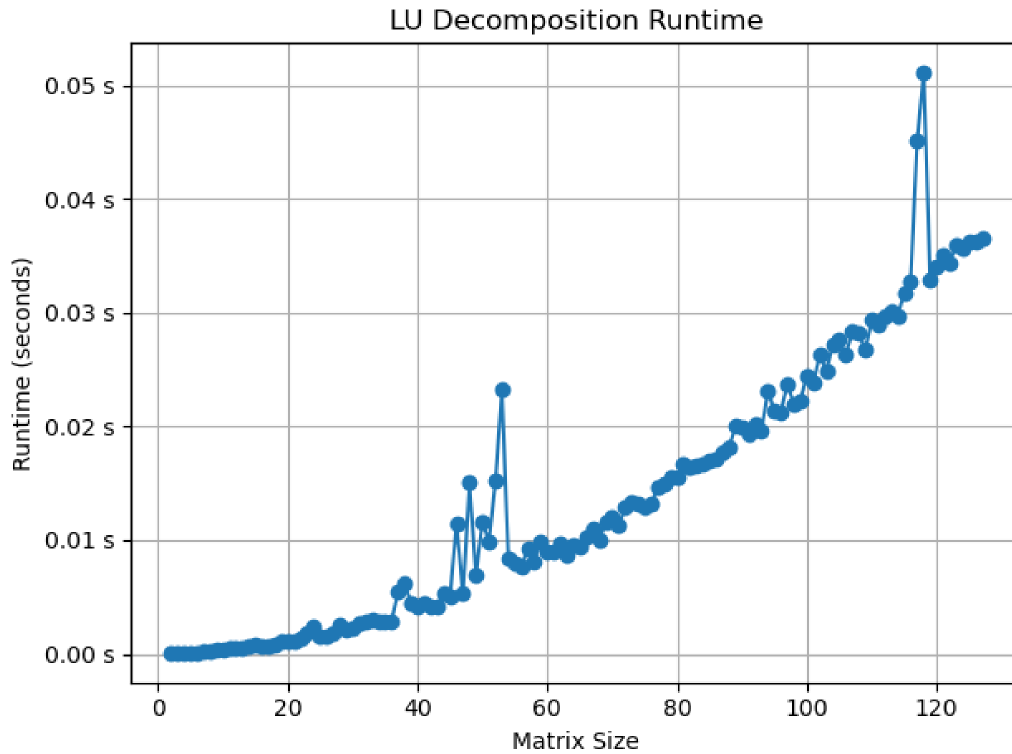

Figure 4: Determinant Calculation Runtime using LU decomposition

The plot above visualizes the runtime of our implementation of LU decomposition to compute the determinant of a matrix, which is notably similar in speed to that of the Bareiss algorithm and significantly faster than the factorial time of Laplace. Because we require n additions and n multiplications in the first column of an nxn matrix, the first column can be expressed as $2n(n-1)$. The second column then requires $n-1$ additions and $n-1$ multiplications, yielding $2(n-1)(n-2)$. Doing this for n number of rows yields the sum of $2(n-i)(n-i+1)$ for $i = 1ton$. This can be expressed as $2/3n^3$, and dropping the constant gives us an in practice time complexity of $O(n^3)$. As we observe in the plot for LU Decomposition, the runtime is relatively consistent and increases linearly with the size of the matrix - with an exception at $size(50)$ and $size(118)$.

4.4 Eigen-decomposition

Time complexity: $O(n^3)$ (usually)

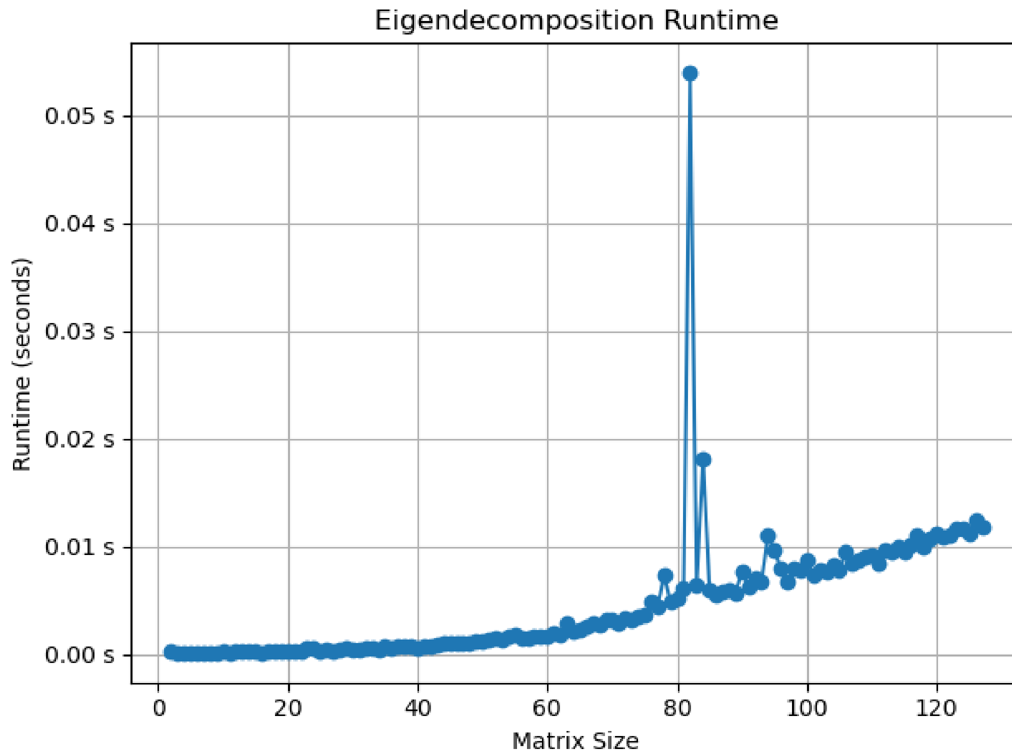

Figure 5: Determinant Calculation Runtime using Eigen-decomposition

Although in theory Eigen-decomposition can be as fast as matrix multiplication (which can be done with$O(nw)$ arithmetic operations), the operations are instead over polynomials of degree n and, as a result, bump the runtime up to $O(n^3)$ in practice. [5] This operation can be done quite efficiently for smaller matrices, but as we increase in size the likelihood of having to do more computationally complex operations involving complex numbers and roots increases, hence the $O(n^3)$ time complexity observed in practice. In the plot for Eigen Decomposition, we observe a spike in runtime at $size(80)$, which is likely due to the increased complexity of the operations involved in the decomposition of a larger matrix.

5 Conclusion

As we can observe and analyze between the display of runtimes for Laplace Expansion, the Bareiss algorithm, LU decomposition, and Eigen-decomposition, as the matrices get arbitrarily larger, all algorithms with a $O(n^3)$ time complexity, all algorithms with the exception of the Laplace Expansion, are considered to be most efficient for calculating the determinant. The Laplace expansion algorithm is superior and the most efficient only when the size of the matrix does not exceed a 5x5 scale. The difference may not be all that much, but it can be noted that the Bareiss expansion is just slightly more efficient compared to that of the LU decomposition and Eigen-decomposition algorithms.

However, please note that these algorithms are not the only ones that can be used to solve for and calculate determinants of matrices. There are other algorithms such as the Strassen algorithm that we have not covered in this paper that have a slightly better runtime ($O(n^{2.807})$) than the ones that we mentioned, which are theoretically known to be better when dealing with more complex and heavy mathematical calculations. However in most cases, the most commonly-used algorithms that we have mentioned, though they may be considered to be less efficient, are still known to be more practical for most computational machines when performing complex mathematical calculations.

References

[1] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning.* Cambridge University Press, 2020.

[2] Felix Limanta. Efficiently calculating the determinant of a matrix. page 1, 2016.

[3] Adrian Mejia. How to find time complexity of an algorithm?, 2020.

[4] soumyadeepdebnath. Big o notation tutorial – a guide to big o analysis, 2024.

[5] Zhao Q. Chen Victor Y. Pan. *The Complexity of the Matrix Eigenproblem.* STOC, 1999.