Efficiently Calculating the Determinant of a Matrix

Felix Limanta 13515065
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515065@std.stei.itb.ac.id
felixlimanta@gmail.com

***Abstract*—There are three commonly-used algorithms to calculate the determinant of a matrix: Laplace expansion, LU decomposition, and the Bareiss algorithm. In this paper, we first discuss the underlying mathematical principles behind the algorithms. We then realize the algorithms in pseudocode Finally, we analyze the complexity and nature of the algorithms and compare them after one another.**

***Keywords*—asymptotic time complexity, Bareiss algorithm, determinant, Laplace expansion, LU decomposition.**

I. Introduction

In this paper, we describe three algorithms to find the determinant of a square matrix: Laplace expansion (also known as determinant expansion by minors), LU decomposition, and the Bareiss algorithm.

The paper is organized as follows. Section II reviews the basic mathematical concepts for this paper, which includes determinants, elementary matrix operations, algorithm complexities, time complexities, and asymptotic time complexities. In Section III, the three methods are discussed in-depth mathematically, whereas in Section IV, the three methods, as well as subroutines necessary to implement them, are implemented in a language-agnostic pseudocode. In Section V, the methods implemented in Section V are analyzed regarding its asymptotic time complexity and other features before being compared to one another. Finally, Section VI concludes the paper while suggesting further improvements.

Terminology and Notation

Unless noted otherwise, all mentions of matrices in this paper are assumed to be square matrices.

To describe iterations mathematically, the notation $i \in [a..b]$ is introduced. The notation is taken from the mathematical concept of number intervals, but instead of describing whether *i* is between *a* and *b* or not, it denotes that the iterator *i* starts from *a* and ends in *b*, incrementing *i* by 1 after each iteration. It is equivalent to the `for` syntax in programming languages, in which $i \in [a..b]$ is equivalent to `for i = a to b`.

The pseudocode described in section III follows the syntax detailed in reference [5].

For convenience, two data types are defined here: `infotype` is either `real` or `integer`, depending on implementation, while `matrix` is an arbitrarily-sized `array of array of infotype`.

II. Basic Mathematical Concepts

A. Determinant

The determinant is a real number associated with every square matrix. The determinant of a square matrix *A* is commonly denoted as det *A*, det(*A*), or |*A*|.

Singular matrices are matrices which determinant is 0. Unimodular matrices are matrices which determinant is 1.

The determinant of a 1×1 matrix is the element itself.

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

The determinant of a 2×2 matrix is defined as the product of the elements on the main diagonal minus the product of the elements off the main diagonal.

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

Finding the determinant of larger matrices will be discussed in later sections.

Some basic properties of determinants are

1. $\det(I) = 1$
2. $\det(A^T) = \det(A)$
3. $\det(A^{-1}) = \frac{1}{\det(A)}$
4. For square matrices A and B of equal size, $\det(AB) = \det(A) \times \det(B)$
5. The determinant of a triangular matrix is the product of the diagonal entries (pivots) $d_1, d_2, \ldots, d_n$.

$$\det(A) = a_{1,1}a_{2,2} \ldots a_{n,n} = \prod_{i=1}^{n} a_{i,i}$$

Determinants are useful for other mathematical subjects. For example, following Cramer's rule, a matrix can be used to represent the coefficients of a system of linear equations, and the determinant can be used to solve the system of linear equations if and only if the matrix is nonsingular. Determinants are also used in determining the inverse of a matrix, where any matrix has a unique inverse if and only if the matrix is nonsingular.

B. Elementary Matrix Operations

Elementary matrix operations play important roles in various matrix algebra applications, such as solving a system of linear equations using and finding the inverse of a matrix.

There are three kinds of elementary matrix operations:

1. Switch two rows (or columns).
2. Multiply each element in a row (or column) by a non-zero number.
3. Add or subtract a row (or column) by multiples of another row (or column).

When these operations are performed on rows, they are called elementary row operations and, likewise, when they are performed on columns, they are called elementary column operations.

Elementary matrix operations have the following effect on determinants.

1. Exchanging two rows or columns from a matrix reverse the sign of its determinant from positive to negative and vice-versa.
$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = -\begin{vmatrix} c & d \\ a & b \end{vmatrix}$$
2. Multiplying one row or column of a matrix by t multiplies the determinant by t.
$$\begin{vmatrix} ta & tb \\ c & d \end{vmatrix} = \begin{vmatrix} ta & b \\ tc & d \end{vmatrix} = t\begin{vmatrix} a & b \\ c & d \end{vmatrix}$$
3. If two rows or two columns of a matrix are equal, or a row or column is the multiply of another row or column, its determinant is zero.
$$\begin{vmatrix} a & b \\ ta & tb \end{vmatrix} = 0$$
4. If $i \neq j$, adding or subtracting t times row i from row j or vice-versa does not change the determinant.
$$\begin{vmatrix} a & b \\ c - ta & d - tb \end{vmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$
5. If A has a row that is all zeros, then $\det(A) = 0$.
$$\begin{vmatrix} a & b \\ 0 & 0 \end{vmatrix} = 0$$

C. Algorithm Complexity

Algorithms are defined as the series of steps necessary to solve a problem systematically. An algorithm, first and foremost, is not only required to be effective and actually solve the problem, but also expected to be efficient. An algorithm might be able to effectively solve a problem, but if the algorithm demands an unreasonable amount of processing time or takes up disproportionately large amount of memory, the algorithm is next to useless.

In application, every algorithm uses up two resources which can be used as measurement: the number of steps taken (time) and the amount of memory reserved to execute those steps (space). The number of steps taken is defined as the time complexity, denoted as T(n), whereas the amount of memory reserved is defined as the space complexity, denoted as S(n).

Matrices are a type of data structure which takes up a relatively large amount of memory, compared to more compact data structures such as linked lists. Therefore, the space complexity of the algorithms discussed later will not be examined in favor of focusing on their time complexity.

D. Time Complexity

The time complexity of an algorithm is measured from the number of instructions executed. The time complexity of an algorithm is *not* measured from the running time of said algorithm because of the differences between every computer's architecture, every programming language and its compiler, and so on. Because of that, the same algorithm implemented in different languages or the same executable run in different machines would yield different running times.

In calculating the time complexity of an algorithm, ideally, all operations are accounted for. For decently complex algorithms, this is all but impossible, so practically, only basic operations are accounted for. For example, the basic operation of a searching algorithm would be the comparison of the searched element with elements in the array. By counting how many comparisons are made, we can obtain the relative efficiency of the algorithm.

The time complexity of an algorithm can be divided into three cases:

- $\mathrm{T_{min}}(n)$: Best case
- $\mathrm{T_{max}}(n)$: Worst case
- $\mathrm{T_{avg}}(n)$: Average case

Example 1. Find the best case, worst case, and average case time complexity of the following sequential search algorithm.

```
function SeqSearch (a: array of infotype,
n: integer, x: infotype) → integer
DICTIONARY
  i: integer
  found: boolean
ALGORITHM
  i ← 1
  found ← false
  while ((i ≤ n) and not(found)) do
    found ← (a[i] = x)
    if not(found) then
      i ← i + 1
  if (found) then
    → i
  else
    → 0
```

- Best case: $a_1 = x \Rightarrow \mathrm{T_{min}}(n) = 1$
- Worst case: $a_n = n$ or n not found $\Rightarrow \mathrm{T_{max}}(n) = n$
- Average case: $\mathrm{T_{avg}}(n) = \frac{1+2+\cdots+n}{n} = \frac{n+1}{2}$

E. Asymptotic Time Complexity

Usually, the exact time complexity of an algorithm is unnecessary. Instead, what is needed is a rough estimate of the time required by the algorithm and how fast that time requirement grow with the size of the input. This is because the efficiency of an algorithm would only be apparent at a large number of inputs. Running the determinant-calculating algorithms on a 3×3 matrix would not yield any notable differences between the algorithm running times. However, if the algorithms are run on a large matrix (say, 5000×5000), differences between running times would be much more apparent.

The asymptotic time complexity is a rough estimate of the time complexity as the number of inputs approaches infinite. The asymptotic time complexity is commonly denoted with the Big-O notation, though other notations exist. Generally, the asymptotic time complexity of an algorithm can be taken from the most significant term in the time complexity function. For example,

$$\mathrm{T}(n) = 192n^5 + 2^n + \log n! = \mathrm{O}(2^n)$$

because for non-negative integer values of n, 2^n contributes the most to the function $\mathrm{T}(n)$.

Table I shows categories of algorithm based on its Big-O notation.

***Table I.** Algorithm categories based on asymptotic time complexity, ordered from smallest to largest*

Complexity	Category
$\mathrm{O}(1)$	constant
$\mathrm{O}(\log n)$	logarithmic
$\mathrm{O}(n)$	linear
$\mathrm{O}(n\log n)$	$n\log n$
$\mathrm{O}(n^2)$	quadratic
$\mathrm{O}(n^3)$	cubic
$\mathrm{O}(2^n)$	exponential
$\mathrm{O}(n!)$	factorial

The top six ($\mathrm{O}(1)$ to $\mathrm{O}(n^3)$) are called polynomial algorithms, while $\mathrm{O}(2^n)$ and $\mathrm{O}(n!)$ are called exponential algorithms. Exponential algorithms grow much faster than polynomial algorithms. For example, if $n = 100$, then $n^3 = 10^6$, whereas $2^n = 1.268\times10^{30}$ and $n! = 9.333\times 10^{157}$.

III. Mathematical Concepts of Determinant Finding Algorithms

A. Determinant Expansion by Minors

Also known as the Laplace Expansion, expansion by minors is a technique to find the determinant of a given square matrix. Although sufficiently efficient for small matrices, other techniques detailed later are much more efficient for very large matrices.

Formally, expansion by minors is defined by

$$\det(A) = \sum_{i=1}^{n} (-1)^{i+j}\, a_{ij} M_{ij}$$

where M_{ij} is the *i, j* minor matrix of A: the determinant of the $(n-1)\times(n-1)$ matrix obtained by deleting the *i*-th row and the *j*-th column of A.

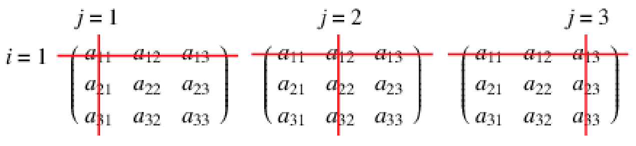

***Figure 1**. Obtaining minor matrices from a larger main matrix*

More informally, expansion by minors can be written as

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{vmatrix} = a_{11}\begin{vmatrix} a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n2} & a_{n2} & \cdots & a_{nn} \end{vmatrix} + \cdots \pm a_{1n}\begin{vmatrix} a_{21} & a_{22} & \cdots & a_{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n(n-1)} \end{vmatrix}$$

Finding the determinant of a 3×3 matrix using expansion by minors can be illustrated with the following example.

Example 2. Find det(A) using expansion by minors, where $A = \begin{bmatrix} 1 & 3 & 5 \\ 0 & 5 & 1 \\ 6 & 5 & 0 \end{bmatrix}$.

Following the above definition, the matrix A can be expanded to minor matrices:

$$\begin{aligned} \det(A) &= 1\begin{vmatrix} 5 & 1 \\ 5 & 0 \end{vmatrix} - 3\begin{vmatrix} 0 & 1 \\ 6 & 0 \end{vmatrix} + 5\begin{vmatrix} 0 & 5 \\ 6 & 5 \end{vmatrix} \\ &= 1\times(-5) - 3\times(-6) + 5\times(-30) \\ &= -137 \end{aligned}$$

Determinant expansion by minors is the simplest method among the three and is commonly taught in class. A derivation of determinant expansion by minors exclusive to 3×3 matrices, called the Sarrus method, is available, although the method cannot be used for larger matrices.

B. LU Decomposition

LU decomposition is a procedure for decomposing a $n\times n$ matrix into a product of a lower triangular matrix L and an upper triangular matrix U.

$$LU = A$$

For a 3×3 matrix, the LU decomposition is as the following.

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{32} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Intuitively, LU decomposition can be performed by applying Gauss elimination, where matrix U is the reduction of matrix A to an upper triangular matrix and matrix L is the multipliers used in each elementary row operation.

More rigidly, let A be a $n\times n$ matrix. First, copy matrix A to matrix U. Iterations denoted by i and j process every element of the matrix, where $i = [1..n]$ and $j = [(i+1)..n]$,

$$l_{ij} = \frac{u_{ji}}{u_{ii}}$$

and

$$u_{jk} = u_{jk} - l_{ij}\times u_{ik}$$

where $k \in [1..n]$.

For calculating determinants, properties 4 and 5 discussed in Section II-1 are used, where $\det(A) = \det(L) \times \det(U)$ and the determinants of L and U are the product of their diagonals.

Example 3. Find det(A) using LU decomposition, where $A = \begin{bmatrix} 1 & 3 & 5 \\ 0 & 5 & 1 \\ 6 & 5 & 0 \end{bmatrix}$.

Using Gauss elimination, matrices L and U can be found to be $L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 6 & -\frac{13}{5} & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 1 & 3 & 5 \\ 0 & 5 & 1 \\ 0 & 0 & -\frac{137}{5} \end{bmatrix}$.
To check if L and U are correct, simply multiply L with U and check if the product matrix is equal to A.

After obtaining L and U, find their determinants by multiplying their diagonals:

$$\det(L) = 1 \times 1 \times 1 = 1$$

$$\det(U) = 1 \times 5 \times \left(-\frac{137}{5}\right) = -137$$

Then, calculate the determinant of A by multiplying the determinants of L and U:

$$\det(A) = \det(L) \times \det(U) = 1 \times (-137) = -137$$

Without a proper ordering, the decomposition may fail as a division by 0 occurs somewhere in the decomposition process and mistakenly imply that matrix A is singular. To rectify this, if pivot $a_{ii} = 0$ for $i \in [1..n]$, simply swap row i with another row j where $a_{ji} \neq 0$, then multiply the entire row by -1 to preserve the determinant value. If this is impossible (a column is composed entirely of 0s), then the matrix is singular and $\det(A) = 0$.

In practice, because the determinant of L is always 1, $\det(A) = \det(U)$. Because of this, in algorithms to calculate determinants using LU decomposition (including the one in this paper), the matrix L is often not computed alongside matrix U and the multipliers used in each elementary row operation is discarded after every iteration.

C. Bareiss Algorithm

The Bareiss algorithm, named after Erwin Bareiss, is an algorithm to calculate the determinant or the echelon form of an integer matrix using only integer arithmetic; that is, any divisions performed are exact (there is no remainder). The method is also used to compute the determinant of a real number matrix and avoids the introduction of rounding errors not present in the input matrix.

Let A be a $n \times n$ matrix. For every iteration denoted by $k \in [1..(n-1)]$, every element in the matrix is processed with the following formula:

$$a_{ij}^{(k+1)} = \frac{1}{a_{(k-1)(k-1)}} \begin{vmatrix} a_{kk}^{(k-1)} & a_{kj}^{(k-1)} \\ a_{ik}^{(k-1)} & a_{ij}^{(k-1)} \end{vmatrix}$$

where $a_{00} = 1$, $i \in [(k+1)..n]$, and $j \in [(k+1)..n]$. The determinant is element $a_{nn}^{(n)}$; that is, the element in the lower right corner after the last iteration is completed.

Example 4. Find det(A) using the Bareiss algorithm, where $A = \begin{bmatrix} 1 & 3 & 5 \\ 0 & 5 & 1 \\ 6 & 5 & 0 \end{bmatrix}$.

Following the algorithm denoted above, we begin by defining $a_{00} = 1$. For the 1st iteration $(k = 1)$, we alter a_{22}, a_{23}, a_{32}, and a_{33}. For $i = 2$ and $j = 2$, we have

$$a_{22}^{\{2\}} = \frac{1 \times 5 - 3 \times 0}{1} = 5.$$

Continuing the process for the remaining elements gives us

$$A^{(2)} = \begin{bmatrix} 1 & 3 & 5 \\ 0 & 5 & 1 \\ 0 & -13 & -30 \end{bmatrix}.$$

For $k = 2$, only a_{33} is changed:

$$a_{33}^{\{3\}} = \frac{5 \times (-30) - 1 \times (-13)}{1} = -137,$$

giving

$$A^{(3)} = \begin{bmatrix} 1 & 3 & 5 \\ 0 & 5 & 1 \\ 0 & -13 & -137 \end{bmatrix}.$$

Thus, $\det(A) = a_{33}^{(3)} = -137$.

As with LU decomposition, division by zero is possible without a proper ordering. Before inputting the matrix into the Bareiss algorithm, swap rows whose pivots are 0, then multiply it by -1.

IV. Implementing Determinant Finding Algorithms in Programs

Note that the snippets in this section assumes that the indices of the matrix start from 1. If the snippets are implemented in languages whose indices start from 0 (such as C), simply decrease iterators by 1.

A. Forming Minor Matrices

The following pseudocode snippet is used to form the minor matrix of a given larger matrix. The function receives the main matrix and integers i and j, which are the respectively, the rows and columns to cross out, and returns the minor matrix formed.

```
function MinorMatrix (M: matrix, input i,
j: integer) → matrix
DICTIONARY
  minor: matrix
  n, a, b, c, d: integer
ALGORITHM
  n ← NRow(M)
  c ← 1
  i traversal [1..n] { Traverse rows }
    if (a ≠ i) then { Skip row i }
      d ← 1
      j traversal [1..n] { Traverse columns }
        if (b ≠ j) then { Skip column j }
          minor[c][d] ← M[a][b]
          d ← d + 1
      c ← c + 1
  → minor
```

The code snippet above traverses the main matrix and copies its values, but skips row `i` and column `j`. It first sets the minor row iterator to 1, then traverses the row of the main matrix. If the loop is not on row `i`, it resets the minor column iterator to 1, then traverses the column of the main matrix. If the loop is not on column `j`, it copies the value from the main matrix to the minor matrix, then increment the minor column iterator. After traversing the column, the minor row iterator is incremented. The resultant minor matrix is then returned.

B. Swapping Rows Whose Pivots are 0

Before implementing LU decomposition and/or Bareiss algorithm in programs, rows whose pivots are 0 must first be swapped in order to avoid a divide by 0 problem.

The following pseudocode snippet describes the algorithm to achieve this. The procedure receives matrix `M` as input, and returns the changed matrix as well as well as a Boolean value denoting whether the matrix is singular or not.

```
procedure SwapRowsWith0Pivot (input/output
M: matrix, output singular: boolean)
DICTIONARY
  n, i, j, k: integer
  temp: infotype
ALGORITHM
  n ← NRow(M)
  singular ← false
  i ← 1
  { Search for 0 pivots }
  while ((i ≤ n) and not(singular)) do
    if (M[i][i] = 0) then
      j ← 1
      { Search for swappable rows }
      while ((j < n) and (M[j][i] = 0)) do
        j ← j + 1
      if (M[j][i] ≠ 0) then { Swap rows }
        k traversal [1..n]
          temp ← M[i][k]
          M[i][k] ← M[j][k]
          M[j][k] ← -temp
      else
        singular ← true
```

The code snippet above traverses the main diagonal to check if any of the matrix' pivots are 0. If there are, it then searches the column where the offending pivot is for a non-zero value. If found, the code then swaps the row where the non-zero value is found with the offending row. Otherwise, if all values of a given column is zero, the matrix is singular and the code terminates.

C. Laplace Expansion

The following pseudocode snippet is used to calculate the determinant of the inputted matrix through Laplace expansion recursively. The function receives the matrix whose determinant shall be calculated and an integer `n`, which represents the number of rows or columns in the matrix, and then returns the determinant of the matrix.

```
function DetLaplace (M: matrix, n: integer)
→ infotype
DICTIONARY
  det: infotype
  minor: matrix
  i, cofactor: integer
ALGORITHM
  if (n = 1) then { Basis, single element matrix }
    det ← M[1][1]
  else { Recurrence }
    det ← 0 { Initialize determinant value }
    cofactor ← -1 { Initialize cofactor }
    i traversal [1..n] { Traverse rows }
      cofactor ← -cofactor { Alternate cofactor }
      minor ← MinorMatrix(M,1,i)
      { Add determinant of current submatrix }
      det ← det + cofactor * M[1,i] *
            DetLaplace(minor,n-1)
  → det
```

The basis of the recursive function is that if the matrix only has a single element, that element is returned as the determinant.

In the recurrence of the function, the determinant is first initialized as 0 and the cofactor as -1. It then traverses the column of the matrix by first negating the cofactor, forming the minor matrix, and finally adding the current value of the determinant with the product of cofactor, the iterated element, and the value returned by the Laplace expansion of the minor matrix. The resultant determinant of the main matrix is then returned.

D. LU Decomposition

The following pseudocode snippet is used to calculate the determinant of the inputted matrix through LU decomposition iteratively. The function receives the matrix whose determinant shall be calculated and then returns the determinant of the matrix.

The code snippet below differs from the mathematical concept previously described in section II. In the concept, the ratios are stored in the matrix L, whereas in the code below, the ratio is single-use; it is only used for that particular iteration. This is because, in practice, the determinant of the matrix L formed through the *LU* decomposition method described in the concept is almost always 1, which makes $\det(A) = \det(U)$ and renders calculating L unnecessary.

```
function DetLU (M: matrix) → real
DICTIONARY
  det, ratio: real
  n, i, j, k: integer
  singular: boolean
ALGORITHM
  SwapRowsWith0Pivot(M,singular)
  if (singular) then { Singular matrix }
    det ← 0
  else { Nonsingular matrix }
    n ← NRow(M)
    i traversal [1..n] { Traverse columns }
      j traversal [(i+1)..n] { Traverse rows }
        ratio ← M[j][i]/M[i][i]
        k traversal [1..n] { Subtract elements }
          M[j][k] ← M[j][k] - ratio *
                    M[i][k]
    det ← 1
    i traversal [1..n] { Multiply pivots }
      det ← det * M[i][i]
  → det
```

The code snippet above can be divided to two parts: transforming the matrix to an upper triangular form and calculating the determinant itself.

Before going to the actual part of LU decomposition, the code first calls `SwapRowsWith0Pivot` to transform the matrix so that there are no zeros in the pivot. If the function fails, the matrix is singular and its determinant is zero. Otherwise, the code continues.

The code then traverses the elements below the main diagonal. It first calculates the ratio needed to gradually zero out elements in the bottom, then traverses the column while subtracting the traversed elements. At the end of the iterations, the matrix would be in an upper triangular form.

In a triangular form, the determinant is easily calculated as the product of the matrix' pivot.

E. Bareiss Algorithm

The following pseudocode snippet is used to calculate the determinant of the inputted matrix through the Bareiss algorithm iteratively. The function receives the matrix whose determinant shall be calculated and then returns the determinant of the matrix.

The code snippet below differs from the mathematical concept previously described in section II. In the concept, the elements are divided with the diagonal element of the previous iteration, whereas in the code below, the variable `pivot` is introduced and used as the divisor. The value of `pivot` starts at 1 and is continually updated in each iteration as the diagonal element of the previous iteration. This is done because in most programming languages, adding `M[0][0]` is unfeasible. If indices start with 1, defining `M[0][0]` would take a considerable amount of space since, in effect, an entire row and column must be added. On the other hand, if indices start with 0, the value of `M[0][0]` is occupied already.

```
function DetBareiss (M: matrix) → infotype
DICTIONARY
  pivot: infotype
  n, i, j, k: integer
  singular: boolean
ALGORITHM
  SwapRowsWith0Pivot(M,singular)
  if (singular) then { Singular matrix }
    det ← 0
  else { Nonsingular matrix }
    n ← NRow(M)
    pivot ← 1
    k traversal [1..(n-1)] { Traverse pivots }
      i traversal [(k+1)..n] { Traverse rows }
        j traversal [(k+1)..n] {Traverse columns}
          { Apply formula }
          M[i][j] ← M[k][k] * M[i][j] -
                    M[i][k] * M[k][j]
          M[i][j] ← M[i][j]/pivot
      pivot ← M[k][k] { Set next pivot }
    det ← M[n][n] { Assign determinant }
  → det
```

Before going to the actual part of LU decomposition, the code first calls `SwapRowsWith0Pivot` to transform the matrix so that there are no zeros in the pivot. If the function fails, the matrix is singular and its determinant is zero. Otherwise, the code continues.

First, `pivot` is initialized to 1. In accordance to the Bareiss algorithm described in the concept, the code traverses the main diagonal of the matrix, then transforms every element to the bottom right of the traversed element through the formula previously described. `pivot` is then updated to the traversed diagonal element. The determinant is the lower-rightmost element of the matrix after the algorithm.

V. Analysis and Comparison

A. Asymptotic Time Complexity Analysis

The following analyses assume that all basic arithmetic operations, assignment, calls, etc. all run in $O(1)$. All time complexities calculated are worst-case scenarios.

For the minor-matrices-forming algorithm, the asymptotic time complexity is as follows.

- Outer loop (row traversal): $O(n)$
- Inner loop (column traversal): $O(n)$

Therefore, the asymptotic time complexity of the algorithm is $O(n \times n) = O(n^2)$.

For the row-swapping algorithm, the asymptotic time complexity is as follows.

- Outer loop (search for 0 pivots): $O(n)$
- 1st inner loop (search for swappable row): $O(n)$
- 2nd inner loop (swap rows): $O(n)$

Therefore, the asymptotic time complexity of the algorithm is $O(n \times \max(n, n)) = O(n^2)$.

For the Laplace expansion algorithm, the asymptotic time complexity is as follows.

- Loop (row traversal): $O(n)$
- Minor matrix forming: $O(n^2)$
- Recursion: $O((n-1)!)$

Therefore, the asymptotic time complexity of the algorithm is $O(n \times \max(n^2, (n-1)!)) = O(n!)$.

For LU decomposition, the asymptotic time complexity is as follows.

- Row swapping: $O(n^2)$
- 1st loop i (column traversal): $O(n)$
- Loop j (row traversal): $O(n)$
- Loop k (subtracting elements): $O(n)$
- 2nd loop i (multiplying pivots): $O(n)$

Therefore, the asymptotic time complexity of the algorithm is $O(\max(n^2, n \times n \times n, n) = O(n^3)$.

For the Bareiss algorithm, the asymptotic time complexity is as follows.

- Row swapping: $O(n^2)$
- Loop i (pivot traversal): $O(n)$
- Loop j (row traversal): $O(n)$
- Loop k (column): $O(n)$

Therefore, the asymptotic time complexity of the algorithm is $O(\max(n^2, n \times n \times n) = O(n^3)$.

B. Comparison

Because the Laplace expansion runs in $O(n!)$, Laplace expansion is only faster than the other two algorithms if the matrix is at most 5×5—at which the difference is too minute to have any significant impact to running time unless the algorithm is performed consecutively on multiple matrices. For even moderately sized matrices, it is significantly outperformed by both the LU decomposition and the Bareiss algorithm.

At a glance, LU decomposition and the Bareiss algorithm performs the same—their asymptotic time complexities are both $O(n^3)$. Performance-wise, they are the same, but LU decomposition has a weakness that makes the Bareiss algorithm better than LU decomposition.

LU decomposition has a weakness in that the ratio needed to transform the matrix to an upper triangular form is not necessarily an integer. Meaning, if the determinant of an integer matrix is to be found using LU decomposition, the resultant upper triangular matrix will most likely be converted to a real number matrix. Furthermore, finding the determinant of a real number matrix is subject to rounding errors due to limitations of information representation, making the resulting determinant slightly less accurate.

Neither the Laplace expansion nor the Bareiss algorithm have the weakness mentioned above; they both preserve the identity of the info type. The Laplace expansion does not use division in its process. The Bareiss algorithm does have division, but it is guaranteed to be exact; that is, in an integer matrix, the division will have no remainder, while in a real number matrix, the division will not significantly change the length of the mantissa.

In conclusion, for arbitrarily large matrices, the Bareiss expansion is slightly superior to the LU decomposition and both the Bareiss algorithm and LU decomposition are vastly superior to Laplace expansion.

VI. Conclusion

For arbitrarily large matrices, the Bareiss expansion is slightly superior to the LU decomposition and both the Bareiss algorithm and LU decomposition are vastly superior to Laplace expansion. The Laplace expansion is superior if and only if the size of the matrix does not exceed 5×5—even then, the difference is only notable if the algorithm is applied on multiple matrices.

Besides the algorithms elaborated in this paper, there are also algorithms to find determinants using fast matrix multiplication (Strassen algorithm with $O(n^{2.807})$ and Coppersmith–Winograd algorithm with $O(n^{2.376})$), which are theoretically better but use heavy mathematical calculations, thus rendering it impractical in most cases. LU decomposition and the Bareiss algorithm, although theoretically worse with $O(n^3)$ complexity, does not use any mathematical calculations more complex than division, making it more practical for all but the most sophisticated machines to use.

Besides real and integer matrices, the aforementioned algorithms can also be expanded to cover complex number or even quaternion matrices, with the resulting determinant being a complex number or quaternion itself.

VII. Acknowledgment

We thank God for His Blessing that enables this paper to be completed on schedule. We also thank Dr. Ir. Rinaldi Munir, M.T. and Dra. Harlili, M.Sc. as the lecturer of IF2121 Discrete Mathematics. We then thank our parent for granting us both material and moral support and guidance in the making of this paper. Finally, we thank our classmates for helping us in class, giving us ideas for this paper, and giving us moral support.

References

[1] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*. Boston, MA: Addison-Wesley Longman, 1974, ch. 6.

[2] Bareiss, E.H., "Sylvester's identity and multistep integer-preserving Gaussian elimination," Math. Comp. 22, 1968.

[3] Jones, J.. (accessed 2016, Dec 3). *The Determinant of a Square Matrix* [Online]. Available: https://people.richland.edu/james/lecture/m116/matrices/determinant.html.

[4] Leggett, D.R, "Fraction-Free Methods for Determinants," M.Sc. thesis, Dept. Math., Univ. Southern Mississippi, Hattiesburg, MS, 2011.

[5] Liem, I., *Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural)*. unpublished.

[6] Munir, R., *Matematika Diskrit*, Ed. 3. Bandung: Teknik Informatika ITB, 2006, ch. 10.
[7] Strang, G., *Introduction to Linear Algebra,* 5th ed. Wellesey: Wellesey-Cambridge Press, 2016, ch. 1.
[8] Weisstein, E.W.. (accessed 2016, Dec 3). *Determinant* [Online]. Available: http://mathworld.wolfram.com/Determinant.html.
[9] Weisstein, E.W.. (accessed 2016, Dec 3). *Determinant Expansion by Minors* [Online]. Available: http://mathworld.wolfram.com/DeterminantExpansionbyMinors.html.
[10] Weisstein, E.W.. (accessed 2016, Dec 3). *LU Decomposition* [Online]. Available: http://mathworld.wolfram.com/LUDecomposition.html.
[11] anonymous. (accessed 2016, Dec 3). *Elementary Matrix Operations* [Online]. Available: http://stattrek.com/matrix-algebra/elementary-operations.aspx.
[12] anonymous. (accessed 2016, Dec 4). *Fastest algorithm for computing the determinant of a matrix?* [Online]. Available: https://stackoverflow.com/questions/27003062/fastest-algorithm-for-computing-the-determinant-of-a-matrix.
[13] anonymous. (accessed 2016, Dec 3). *Lecture 12 – LU Decomposition* [Online]. Available: http://www.math.ohiou.edu/courses/math3600/lecture12.pdf.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Desember 2016

[illegible]

Felix Limanta 13515065