

# Calculating Determinants of Symbolic and Numeric Matrices

## *A Starting Point*

**Technical Report by  
Brent M. Dingle**

**Texas A&M University  
May 2004, November 2005**

### **Abstract:**

There have been many papers published on how to calculate the determinant of a matrix with symbolic entries. Most of the methods used for numerical matrices will work for symbolic matrices, however they are slow. To assist the young computer scientist in discovering this, the more common definitional methods are presented here along with C++ source code. Thus this paper should serve as a place to start developing a system that could calculate determinants of symbolic matrices, while at the same time making the difficulties of performing such a task more obvious than just listening to a theoretical discussion of the more advanced techniques. In sum, this paper will present several methods of determinant calculation with details on their implementation. This is done with the intent to make it easier for the beginning student to become familiar with the problems surrounding this task.

# 1 Introduction

This paper will begin with three definitions of determinant. Each definition is equivalent to any other, however each offers a different perspective on determinant. The first definition is for abstract thought, the other two definitions lead to methods of calculating the determinant. The third section will present a brief discussion on solving systems of equations, which leads to the Gaussian elimination method for calculating the determinant. It is not until section 4 that we even mention symbolic determinants, and it is in this section the actual methods for calculating such will be described. The C++ source code for implementing the key components of each method will also be found in this section. In section 5 we offer a brief summary of speeds and scenarios for the usage of each method.

## 2 Determinant Definition

It should be understood that in terms of this paper the determinant is only defined for square matrices. If a matrix is not square then its determinant does not exist. With that in mind we begin with the geometric definition of determinant and progress to the classical algebraic definition of determinant. These definitions apply whether the matrix has numerical or symbolic entries.

### 2.1 The Geometric Definition

The most intuitive definition of determinant is the geometric definition. It is this definition that is often overlooked and rarely used for computation. We mention it here for completeness and in the hope that a visual picture may aid in the understanding and usage of the determinant.

We will begin with a simple 1x1 matrix. In this case the determinant of the matrix is the signed length of the line from the origin to the point on the number line marked by the entry of the matrix. So if the single entry of the matrix is positive, we consider the determinant to be the length of the line from the origin to the point going in the positive 'x' direction. If the entry is negative then the determinant is the negative of the length of the line from the origin to the point going in the negative 'x' direction.

In the case of a 2x2 matrix we look at the matrix as a set of two points in the Euclidean plane. Using these two points we make a parallelogram that includes the origin. The determinant is then the signed area of the parallelogram.

For example if the matrix was:  $\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$  then we would have a rectangle with corner points at (0, 0), (2, 0), (2, 1) and (0, 1). And the determinant would be (positive) 2.

For a 3x3 matrix the concept is much the same. We consider the matrix to be 3 points in 3-dimensional Euclidean space. We create a parallelepiped that includes the three points and the origin. The determinant is then the signed volume of the parallelepiped.

This concept extends to the higher dimensions of Euclidean space. So the determinant of an  $n \times n$  matrix would be the 'volume' of the  $n$ -dimensional parallelepiped formed from the  $n$  points of the matrix.

It should be clear that this geometric definition of determinant fails to offer an obvious method to calculate the determinant. However, it may be useful in thinking about it.

## 2.2 The Classical Algebraic Definition

For the classical definition of determinant we must first define permutation: Given a set  $S_n = \{ i, i = 0 \text{ to } n-1 \}$  a rearrangement of the elements of  $S_n$  is a permutation of  $S_n$ . For example, let  $S_3 = \{ 0, 1, 2 \}$ . Then 012, 021, 102, 120, 201, 210 are the six permutations of  $S_3$ . Notice for any  $S_n$  there will always be  $n!$  possible permutations.

Another important thing to notice in each permutation is the number of inversions. A pair of elements  $(p_i, p_j)$  is called an inversion in a permutation if  $i > j$  and  $p_i < p_j$ , or rather  $p_i$  comes before  $p_j$ . So in the above example let 012 be the original ordering. Then the following are true:

012 has 0 inversions	021 has 1 inversion (the 21)
102 has 1 inversion (the 10)	120 has 2 inversions (10 and 20)
201 has 2 inversions (21 and 20)	210 has 3 inversions (21, 20 and 10)

In a given permutation  $j_0 j_1 j_2 \dots j_{n-1}$  of  $S_n$  the permutation is called an odd permutation if the number of inversions is odd. The permutation is called an even permutation if the number of inversions is even.

And finally we arrive at the definition of determinant. Let  $A$  be an  $n \times n$  matrix. Such that

$$A = [a_{ij}] = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

The determinant of  $A$  is defined as

$$\det(A) = |A| = \sum (\pm) a_{0,j_0} a_{1,j_1} a_{2,j_2} \cdots a_{n-1,j_{n-1}} \quad \text{over all permutations of } S_n$$

where  $j_0 j_1 j_2 \dots j_{n-1}$  is a permutation of  $S_n$   
 and  $(\pm) = +$  if  $j_0 j_1 j_2 \dots j_{n-1}$  is an even permutation  
 and  $(\pm) = -$  if  $j_0 j_1 j_2 \dots j_{n-1}$  is an odd permutation.

Notice if we follow this definition there will be six terms in the summation for a determinant of a  $3 \times 3$  matrix, there will be 24 terms in the summation for a  $4 \times 4$  matrix, there will be..., there will be  $n!$  terms in the summation for an  $n \times n$  matrix.

## 2.3 The Recursive Definition using Minors and Cofactors

This definition is also a method, often referred to as “expansion about the minors.” However, in Computer Science this method of derivation is also a recursive definition. This method requires a couple more definitions and an explicit statement of the determinant of a 2x2 matrix. We will begin with the definition of a minor:

A minor of a given element in a matrix is the determinant that results from the matrix created by deleting the row and column of the given element. For example, consider the following.

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

The minor of  $a_{0,0}$  is obtained by deleting column 0 and row 0 and taking the determinant of the resulting matrix, thus

$$\text{the minor of } a_{0,0} = \text{determinant of } \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$$

Likewise

$$\text{the minor of } a_{0,1} = \text{determinant of } \begin{bmatrix} a_{1,0} & a_{1,2} \\ a_{2,0} & a_{2,2} \end{bmatrix}$$

And

$$\text{the minor of } a_{0,2} = \text{determinant of } \begin{bmatrix} a_{1,0} & a_{1,1} \\ a_{2,0} & a_{2,1} \end{bmatrix}$$

For our purposes of calculating the determinant the above 3 minors would be sufficient, though you could also take the minors of  $a_{1,0}$  or  $a_{2,1}$  or any of the other elements in a similar fashion. Also notice for the purposes of determinant calculation if we had started with a 4x4 matrix we would have had 4 minors, each being a determinant of a 3x3 matrix. Similarly a 5x5 matrix would have had 5 minors, each being a determinant of a 4x4 matrix and so on. Thus an  $n \times n$  matrix would have  $n$  minors, each being a determinant of an  $n-1 \times n-1$  matrix.

With the definition of minor in place we now define a cofactor of a given element in a matrix to be the minor or negation of the minor of the given element, depending upon the element's location in the matrix. If the row and column of the element add up to be an even number then the cofactor is the minor, otherwise the cofactor is the negation of the minor. Thus in the example above, the cofactor of  $a_{0,0}$  would be the minor of  $a_{0,0}$ . But the cofactor of  $a_{0,1}$  would be the negation of the minor of  $a_{0,1}$ .

We will now define the determinant of a 2x2 matrix to be as follows:

$$\text{Let } A = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \text{ then } |A| = a_{0,0} * a_{1,1} - a_{0,1} * a_{1,0}.$$

With this definition we may now define the determinant of any  $n \times n$  matrix. Let  $A$  be an  $n \times n$  matrix then the determinant of  $A$  is defined as:

$$\det(A) = \sum_{j=0}^{n-1} a_{0,j} \cdot \text{cofactor}(a_{0,j})$$

This definition will recurse down until the definition of the determinant of a 2x2 matrix can be applied. While this would seem easy to implement it has problems, most of which are shared among all recursive solutions.

### 3 Solving Systems of Equations

In this section we move to a related topic of solving systems of equations. This is often done using matrix representations and operations. In fact it is common to see a system of  $n$  equations and  $n$  unknowns written in the form of  $Ax = b$ . Where  $A$  is an  $n \times n$  matrix and  $x$  and  $b$  are  $n \times 1$  vectors, where  $x$  is usually a vector of variable names and  $b$  is usually a vector of numeric values. Thus assuming there is a solution  $x$  we could find it by saying  $x = A^{-1}b$ . It is here that determinants come into play. If the determinant of  $A$  is not zero, then  $A^{-1}$  exists and there is a solution. From this it would be obvious that if we have a method to find the solution of  $Ax = b$  then we might have a way to find the determinant of  $A$  as well. This is indeed the case.

A common way to solve the system  $Ax = b$  is to use matrix operations to transform  $A$  into an upper (or lower) triangular matrix and then back solve. This is also referred to as Gaussian elimination. The beauty of this method is that it also allows for the easy calculation of the determinant as it has been proven the determinant of an upper (or lower) triangular matrix is just the product of the diagonal elements, which by the recursive definition of determinant should be obvious.

It should be noted here that for every row or column swap required to achieve the upper triangular form, the determinant will be off by a multiply of negative one. More specifically if in transforming  $A$  into an upper triangular matrix there were an odd number of row and column swaps then the determinant is the negation of the product of the diagonal elements. If there were an even number of row and column swaps then the determinant is exactly the product of the diagonal elements.

## 4 Methods of Finding the Determinant

### 4.1 Applying the Classical Method

One of the most straightforward ways to find the determinant is the direct application of the classical algebraic definition as stated in section 2.2. Using this method all that needs to be done is to calculate the product of every possible permutation and then sum them.

The algorithm would go something like:

```
CalcDetClassic(input Matrix, output Determinant)
{
    Check for invalid conditions (not square etc)
    If number of rows = 1 then return only element, e[0][0]
    If number of rows = 2 then return e[0][0]*e[1][1] - e[0][1]*e[1][0]

    Initialize a permutation vector
    Initialize return value to zero
    While not used all permutations
    {
        Calculate the product of the current permutation
        If the current permutation is even
            Add the product to the return value
        Else
            Subtract the product from the return value

        Get the next permutation
    }
}
```

Notice this algorithm will work regardless of whether the matrix is composed of numbers or symbolic polynomials. For it to function with polynomials the data structure representing them would need to support addition, subtraction and multiplication.

The obvious problem with this method is the time requirement. There will be  $n!$  terms that must be added together. That means there are at least  $n!$  multiplies and additions and it is quite likely more multiplies than that. This is bad if you are dealing with just numbers but terrible if you are working with polynomials. Consider if you must multiply  $(x + 3) * (y - 4)$ . While this is “one” multiply it actually requires four multiplies and an addition.

The C++ source code for this can be found in Appendix A.

## 4.2 A Recursive Method

This method is perhaps the easiest to implement. However for it to be useful some care must be taken in memory allocation and depth of recursion. The algorithm would be:

```
CalcDetRecurse(input Matrix, output Determinant)
{
    Check for invalid conditions (not square etc)
    If number of rows = 1 then return only element, e[0][0]
    If number of rows = 2 then return e[0][0]*e[1][1] - e[0][1]*e[1][0]

    Ret_val = 0
    For j = 0 to (number of columns - 1)
    {
        SubMatrix = Matrix with row 0 and column j removed
        Cofactor = CalcDetRecurse(SubMatrix, Cofactor)
        If j is odd
            Then Cofactor = -1 * Cofactor
        Ret_val = Ret_val + e[0][j] * Cofactor
    }
    Return Ret_val
}
```

This method will also work for numerical or symbolic matrices. However it too will have a large number of multiplies, and in the case of polynomials that is not such a good thing. In particular notice that in the for-loop there will be  $n$  multiplies and  $n$  additions and there will be  $n$  recursive calls where each call will have  $n-1$  multiplies and  $n-1$  additions which in turn will have another  $n-1$  recursive calls that each make  $n-2$  multiplies and  $n-2$  additions and so on until  $n-i$ , is two.

So in the case of  $n = 3$  there would be three multiplies and three additions in the initial call and three recursive calls each having two multiplies and one subtraction so the total cost would be  $3 + 3 * 2 = 9$  multiplies and  $3 + 3*1 = 6$  additions. In the case of  $n = 4$  there would be four multiplies and four additions with four recursive calls each having nine multiplies and six additions, so the total cost would be  $4 + 4 * 9 = 40$  multiplies and  $4 + 4 * 6 = 30$  additions. In the case of  $n = 5$  there would be five multiplies and five additions with five recursive calls each having 40 multiplies and 30 additions, for a total cost of  $5 + 5 * 40 = 205$  multiplies and  $5 + 5 * 30 = 155$  additions. For the case of  $n = 6$ , there would be six multiplies and six additions with six recursive calls each having 205 multiplies and 155 additions, the total cost would end up being  $6 + 6 * 205 = 1236$  multiplies and  $6 + 6 * 155 = 936$  additions. Notice this is actually worse than  $n!$  multiplies and  $n!$  additions.

Fortunately there is a way to improve this which will be discussed in Section 4.4. The C++ source code for this, which implements the improvement in section 4.4, can be found in Appendix B.

### 4.3 Gaussian Method using an Upper Triangular Form

For this method we simply transform the original matrix into an upper triangular matrix. As we transform the matrix we keep track of how many row and column swaps we perform. We then calculate the determinant by taking the product of the diagonal elements. If the number of row and column swaps was odd we multiply the result by negative one. If the number was even we do nothing else. The key component of this method is the function which converts the original matrix into an upper triangular one. This can be done using the following algorithm:

```
MakeUpper(input/output Matrix, output Number of row/column swaps)
{
    If number of rows = 1 then return

    swap_count = 0
    For i = 1 to (number of rows - 1)
    {
        For k = 0 to (i - 1)
        {
            factor = e[i][k] / e[k][k]
            For j = (k + 1) to (number of columns - 1)
            {
                e[i][j] = e[i][j] - (factor * e[k][j])
            }
        }
        // might have set diagonal element to zero, requiring a pivot
        if e[i][i] == 0
            then swap column i for some column j where e[i][j] != 0
                and increase swap_count by 1
    }
    if swap_count is odd return -1 else return 1
    // notice the lower triangle of elements is not explicitly zeroed
    // that could be added in
}
```

This method, like the previous ones, will work for any type of matrix. This method requires approximately  $O(n^3)$  multiplications, divisions and additions. It is probably the most often implemented method of calculating determinants as it is the method most people are taught in linear algebra classes. Notice for comparison to the previous methods for  $n=3$ ,  $3^3 = 27$ , for  $n=4$ ,  $4^3 = 64$ , for  $n=5$ ,  $5^3 = 125$ , for  $n=6$ ,  $6^3 = 216$ , and so on. When working with polynomials the division portion of this algorithm is often the most costly, it would be nice if the number of divisions could be significantly reduced, or at least guaranteed to ‘come out evenly’ every time. Which is one of the motivations for the algorithm in the next section.

The C++ source code for this can be found in Appendix C.

## 4.4 Improved Recursive Method

This method uses both recursion and an elimination trick to assist in the calculation of the determinant. This method is based on one proposed in [Bare1968] it is sometimes referred to as a fraction free determinant calculation. The concept is to create an upper triangular matrix but to also keep track of the determinant as we go. The algorithm goes something as follows:

```

CalcDetRecurse(input Matrix, output Determinant)
{
  Check for invalid conditions (not square etc)
  If number of rows = 1 then return only element, e[0][0]
  If number of rows = 2 then return e[0][0]*e[1][1] - e[0][1]*e[1][0]

  // Construct SubMatrix (with 1 less row and column than this Matrix)
  For i = 1 to (number of rows - 1)
  {
    For j = 1 to (number of rows - 1)
    {
      subtract_me = e[i][0] * e[0][j]
      e[i][j] = e[i][j] * e[0][0] - subtract_me
    }
  } // Submatrix is e[1][1] to e[n-1][n-1] inclusive

  Ret_val = CalcDetRecurse(SubMatrix, Determinant)
  For i = 1 to num_rows - 2
  {
    Determinant = Determinant / e[0][0]
  }
}

```

Notice this algorithm appears to run with about  $O(n^2)$  multiplications and additions, however it is  $n^2 + (n-1)^2 + \dots = (n-1)(2n^2 + 5n + 6) / 6 = O(n^3)$ . Which is still the best so far. It will also work with any type of matrix. However it is slightly complicated. To understand this method it is best to work through an example. We will begin with a numerical 4x4 matrix and show each submatrix created.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 3 & 7 & 1 \\ 3 & 2 & 2 & 5 \\ 13 & 11 & 3 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} -7 & -8 & -19 \\ -4 & -7 & -7 \\ -15 & -36 & -46 \end{bmatrix} \rightarrow \begin{bmatrix} 17 & -27 \\ 132 & 37 \end{bmatrix} \rightarrow 4193$$

And then as the recursion unwinds:  $4193 / -7 = -599 \rightarrow -599 / 1 = -599, -599 / 1 = -599$

Now to fully realize why the method works it might be a good idea to look at a symbolic matrix and notice how things cancel and why they cancel. To demonstrate this we will offer a 4x4 matrix and reduce it to an upper triangular matrix. The notation will be as follows: the subscripts will remain the same and the superscripts will denote the iteration, the lack of a superscript means iteration 1.

So let us begin with the following matrix:

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \\ d_0 & d_1 & d_2 & d_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ 0 & b_1 - a_1 \frac{b_0}{a_0} & b_2 - a_2 \frac{b_0}{a_0} & b_3 - a_3 \frac{b_0}{a_0} \\ 0 & c_1 - a_1 \frac{c_0}{a_0} & c_2 - a_2 \frac{c_0}{a_0} & c_3 - a_3 \frac{c_0}{a_0} \\ 0 & d_1 - a_1 \frac{d_0}{a_0} & d_2 - a_2 \frac{d_0}{a_0} & d_3 - a_3 \frac{d_0}{a_0} \end{bmatrix}$$

$$\text{Let } b_1^2 = b_1 - a_1 \frac{b_0}{a_0} = (a_0 b_1 - a_1 b_0) / a_0$$

$$b_2^2 = b_2 - a_2 \frac{b_0}{a_0} = (a_0 b_2 - a_2 b_0) / a_0$$

and so on.

Notice it is the latter form that is being used in this section's algorithm. Continuing on:

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ 0 & b_1^2 & b_2^2 & b_3^2 \\ 0 & c_1^2 & c_2^2 & c_3^2 \\ 0 & d_1^2 & d_2^2 & d_3^2 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ 0 & b_1^2 & b_2^2 & b_3^2 \\ 0 & 0 & c_2^2 - b_2^2 \frac{c_1^2}{b_1^2} & c_3^2 - b_3^2 \frac{c_1^2}{b_1^2} \\ 0 & 0 & d_2^2 - b_2^2 \frac{d_1^2}{b_1^2} & d_3^2 - b_2^2 \frac{d_1^2}{b_1^2} \end{bmatrix}$$

Again performing a renaming and another iteration we arrive at:

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ 0 & b_1^2 & b_2^2 & b_3^2 \\ 0 & 0 & c_2^3 & c_3^3 \\ 0 & 0 & d_2^3 & d_3^3 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ 0 & b_1^2 & b_2^2 & b_3^2 \\ 0 & 0 & c_2^3 & c_3^3 \\ 0 & 0 & 0 & d_3^3 - c_3^3 \frac{d_2^3}{c_2^3} \end{bmatrix}$$

The straightforward Triangular (Gaussian) method would then calculate the determinant by multiplying:  $a_0 b_1^2 c_2^3 \left( d_3^3 - c_3^3 \frac{d_2^3}{c_2^3} \right)$ . However, the recursive method described in this section notices that there is automatically a large amount of cancellation. Specifically it can be shown,  $c_2^3 \left( d_3^3 - c_3^3 \frac{d_2^3}{c_2^3} \right)$  can be evenly divided by  $b_1^2$  exactly  $3-2 = 1$  time and the quotient of that division can in turn be evenly divided by  $a_0$  exactly  $4-2 = 2$  times.

## 4.5 Method using an Interpolation Trick (univariate)

This particular method has a variety of implementations and clever performance enhancements. For this paper we will limit things to be as basic as possible. This method is specifically designed for a matrix that has entries which are univariate polynomials. In fact there can only be one variable, though it can be found in multiple entries in the matrix. The motivation for this method is that numeric computations can be performed much faster than symbolic computations – they have the advantage of hardware and compiler optimization techniques. So if we put a number in for the variable we will have a strictly numeric matrix for which we can quickly find the determinant.

To begin the reasoning of this method, notice that the determinant if solved symbolically would be a univariate polynomial equation. If we can predetermine the degree of this equation to be  $d$  we can use  $d+1$  values for the variable and calculate the determinant  $d+1$  times. This would give us  $d+1$  points to use to interpolate what the univariate polynomial equation would be. This of course assumes we have a fast way to calculate numeric determinants and a fast interpolation method, both of which can readily be found, for example in [Pres2002].

### 4.5.1 Determining the Degree of the Determinant

So the actual problem to solve is finding  $d$ , the degree of the resulting determinant. A method to do this is presented in [Henr1999] which in turn was based on a method proposed in [Door1979]. While the method described in those papers is effective we will present a simpler method, which may or may not be as efficient. It is based on the improved recursive method described above. The idea is as follows:

Given a symbolic  $n \times n$  matrix, create a new  $n \times n$  matrix where each entry  $e[i][j]$  is the highest degree of the variable appearing in the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column of the original matrix. Follow the general steps of the Improved Recursive Algorithm of section 4.4, however all we need to do is keep track of the degree of the variable that would result from the various multiplications. Notice it is possible that this highest degree may get cancelled in a subtraction, however we will assume this does not happen, and thus will arrive at a maximum bound on the degree. Further note that the divisions could give us a minimum bound on the degree, as all the divisions must come out even. It should be obvious this calculation will take no more time than required to calculate one numerical determinant.

The actual algorithm would go something as follows, we assume the input matrix is the matrix of maximum degrees:

```
CalcDegreeOfDet(input Matrix, output MaxDeg)
{
    Check for invalid conditions (not square etc)
    If number of rows = 1
        then return only element, e[0][0]

    If number of rows = 2
        then return Max(e[0][0]+e[1][1], e[0][1]+e[1][0])
}
```

```

// Construct SubMatrix (with 1 less row and column than this Matrix)
For i = 1 to (number of rows - 1)
{
  For j = 1 to (number of rows - 1)
  {
    subtract_me = e[i][0] + e[0][j]
    e[i][j] = Max( e[i][j] + e[0][0], subtract_me)
  }
} // Submatrix is now e[1][1] to e[n-1][n-1] inclusive
MaxDeg = CalcDetRecurse(SubMatrix, MaxDeg)

For i = 1 to num_rows - 2
{
  MaxDeg = MaxDeg - e[0][0]
}

Return MaxDeg
}

```

#### 4.5.2 Examples of Degree Calculation

To illustrate the above algorithm consider the following:

$$\text{Let } A = \begin{bmatrix} x & x^2 & 5 \\ x^3 & 7 & x \\ x^2 & x^2 & x^4 \end{bmatrix}, \text{ then the matrix of maximum degrees is } \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 1 \\ 2 & 2 & 4 \end{bmatrix}.$$

Applying the algorithms we get the following:

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 1 \\ 2 & 2 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} \max(0+1, 2+3) & \max(1+1, 3+0) \\ \max(2+1, 2+0) & \max(4+1, 2+1) \end{bmatrix} \rightarrow \begin{bmatrix} 5 & 3 \\ 3 & 5 \end{bmatrix} \rightarrow \max(10, 6) = 10$$

Unwinding the recursion we subtract 1 from 10 exactly  $3-2 = 1$  time, for a result of  $d = 9$ . This is correct as the determinant is  $-x^9 + 13x^5 - x^4 - 35x^2$ .

Another example is as follows.

$$\text{Let } A = \begin{bmatrix} x^3 & x^2 & 5 & x \\ x^2 & x^3 & 7 & x \\ 1 & x^2 & x^2 & x^4 \\ 3x & 5 & 9x^2 & 2 \end{bmatrix}, \text{ then the matrix of maximum degrees is } \begin{bmatrix} 3 & 2 & 0 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 2 & 2 & 4 \\ 1 & 0 & 2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 2 & 0 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 2 & 2 & 4 \\ 1 & 0 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 3 & 4 \\ 5 & 5 & 7 \\ 3 & 5 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 11 & 13 \\ 11 & 12 \end{bmatrix} \rightarrow 24$$

Unwinding we see that  $24 - 6 = 18$ , and then  $18 - 3 - 3 = 12$ . So the degree of the determinant should be no more than 12. And in fact the degree is exactly 12 as the determinant is  $-9x^{12} + 9x^{10} + 26x^8 + 2x^7 - 20x^6 - 18x^5 + 16x^4 - 10x^3 + 14x^2 - 10x$ .

### 4.5.3 Calculating the Interpolation Points

Once we have determined the degree of the determinant to be  $d$  we will need to calculate the determinant at  $d + 1$  unique values. To illustrate this consider the first example in

section 4.5.2 where  $A = \begin{bmatrix} x & x^2 & 5 \\ x^3 & 7 & x \\ x^2 & x^2 & x^4 \end{bmatrix}$ . We found the degree of this determinant to be 9,

so we need to evaluate the determinant for  $9+1 = 10$  unique values of  $x$ . For notation purposes let  $|A(v_i)|$  denote the determinant of  $A$  when a value of  $v_i$  is placed in for  $x$ . Let  $v_i = i$  for  $i = -4$  to  $5$ , thus giving us 10 values =  $\{-4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$ .

Using whatever fast numerical determinant method we like, we find that

$$\begin{aligned} |A(v_{-4})| &= 248016 \\ |A(v_{-3})| &= 16128 \\ |A(v_{-2})| &= -60 \\ |A(v_{-1})| &= -48 \\ |A(v_0)| &= 0 \\ |A(v_1)| &= -24 \\ |A(v_2)| &= -252 \\ |A(v_3)| &= -16920 \\ |A(v_4)| &= -249648 \\ |A(v_5)| &= -1914000 \end{aligned}$$

We then use whatever fast numerical interpolation routine we like to find the polynomial that goes through the points:

$$\begin{aligned} (-4, 248016), & & (1, -24) \\ (-3, 16128), & & (2, -252) \\ (-2, -60), & & (3, -16920) \\ (-1, -48), & & (4, -249648) \\ (0, 0), & & (5, -1914000) \end{aligned}$$

And arrive at the answer of:  $-x^9 + 13x^5 - x^4 - 35x^2$ .

## 5 Summary

From the above the reader should now have a basic understanding of what is required to calculate determinants. It should be apparent that the obvious (definitional) methods, while useful in solving small matrices may not be the optimal solution for larger problems. With run times in the order of  $O(n!)$  and  $O(n^3)$  things will take a while to run, even when using just numeric matrices. When polynomial entries are allowed run times become even worse in implementation [Gent1973].

For applications dealing with symbolic matrices, from our experience, currently, the classical method can solve problems up to  $n = 9$ , within several minutes. The Gaussian based method can solve problems up to size about  $n = 6$ . The improved recursive method can likewise solve such problems up to a size of about  $n = 12$ . All of them are capable of solving larger problems, within restrictions of maximum values held within data types, however the time required quickly becomes unreasonable. The interpolation method can solve larger problems but its effectiveness depends greatly on the degree of the determinant polynomial.

One thing to learn from these methods would be that the hybridization of the methods is likely to improve the effective runtimes. Specifically if a small amount of analysis is done prior to selecting a method to calculate the determinant, it might make things extremely easy. Likewise one method might be used to begin the determinant calculation and another used to finish the smaller, submatrix problems. If automated techniques could be developed to perform such tasks then runtimes might improve. With this statement it should be understood that there currently are many specialized techniques for a great many problems, an integration or a useful generalization of these techniques should be done in the future. It is hoped that this paper might prove to be helpful for such an endeavor.

In the Appendices of this paper you will find source code implementing most of the algorithms described in this paper. The source code is based on polynomial classes as described in "Designing a Multivariate Polynomial Class" by Brent M. Dingle, April 2004.

## Bibliography

- [Bare1968] Erwin H. Bareiss, "Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination," *Mathematical Computation* 22, 103, pp. 565 – 578, 1968.
- [Dodg1867] C. L. Dodgson, *An Elementary Treatise on Determinants, with Their Application to Simultaneous Linear Equations and Algebraical Geometry*. London: Macmillan, 1867.
- [Door1979] P. M. Van Dooren, P. Dewilde and J. Vandewalle, "On the Determination of the Smith-MacMillan Form of a Rational Matrix From Its Laurent Expansion," *IEEE Transactions on Circuits and Systems*, Vol. 26, No. 3, pp. 180-189, 1979.
- [Gent1973] W. M. Gentleman and S. C. Johnson, "Analysis of algorithms, a case study: Determinants of polynomials," *Proceedings of the fifth annual ACM symposium on Theory of computing*, ACM Press, pp. 135-141, 1973.
- [Henr1999] D. Henrion and M. Sebek, "Improved Polynomial Matrix Determinant Computation." *IEEE Trans. on CAS - Pt I. Fundamental Theory and Applications*, Vol. 46, No. 10, pp. 1307-1308, October 1999.
- [Pres2002] William H. Press (Editor), Saul A. Teukolsky (Editor), William T. Vetterling (Editor), Brian P. Flannery (Editor), Numerical Recipes in C++ 2<sup>nd</sup> edition, Cambridge University Press, February 2002.

## Appendix A – Source Code for Classical Method

```
long CPolyMat::CalcDetClassic(CPolyfrac &answer_polyf)
{
    CPolyfrac tmp_polyf;
    std::vector<int> v;    // contains indices of permutations
    bool even, more;
    int i;
    long num_terms, cur_term;
    long ret_val;

    // default to failure
    ret_val = -1;
    answer_polyf.SetInteger(0);

    if (m_NumRows != m_NumCols) { return ret_val; } // nonsquare = det undefined
    if (m_NumRows <= 0) { return ret_val; }        // no matrix entries

    if (m_NumRows == 1)    // Do case 1 by 1
    {
        answer_polyf = m_Mat[0][0];
        ret_val = 1;
    }

    else if (m_NumRows == 2) // Do case 2 by 2
    {
        answer_polyf = m_Mat[0][0] * m_Mat[1][1];
        tmp_polyf = m_Mat[0][1] * m_Mat[1][0];
        answer_polyf -= tmp_polyf;
        ret_val = 1;
    }

    else
    {
        v.push_back(0); // initialize permutation index vector v[]
        num_terms = 1;
        for (i=1; i < m_NumRows; i++)
        {
            v.push_back(i);
            num_terms *= i;    // num_terms = (n-1)! = (n-1)*(n-2)*...*2*1
        }

        num_terms *= m_NumRows;

        if (num_terms <= 0)
        {
            PostError();
            ret_val = -1;
            return ret_val;
        }

        // answer_polyf init'd to zero above
        even = true;
        more = false;
        for (cur_term=0; cur_term < num_terms; cur_term++)
        {
            NextPermute(v, m_NumRows, &more, &even);

            tmp_polyf = m_Mat[0][ v[0] ] * m_Mat[1][ v[1] ];
            for (i=2; i < m_NumRows; i++)
            {
                tmp_polyf *= m_Mat[i][ v[i] ];
            } // end for i

            if (even)
            {
                answer_polyf += tmp_polyf;
            }
            else
            {
                answer_polyf -= tmp_polyf;
            }
        }
    }
}
```

```

    }
    } // end for cur_term

    ret_val = 1;
} // end case 3 by 3 or greater

return ret_val;
} // end CalcDetClassic

```

Notice that the function NextPermute() computes all the permutations of N integers, one at a time. When the function is first called more should be set to false, so the function will return the 'original' permutation. The parameters should be obvious in meaning, the *n* is the number of objects being permuted, *v*[] is the permutation, *more* is a flag variable and *even* will be set to true or false depending on if the returned permutation is even or odd. It is assumed that if the initial permutation is 0, 1, 2, 3 then the second will be 0, 1, 3, 2 and the third will be 0, 2, 1, 3 and so on. Effectively all the permutations starting with 0 are done first, then all those starting with 1, then all those starting with 2, and so on. The source code would look something like:

```

void CPolyMat::NextPermute(std::vector<int> &v, long n, bool *more, bool *even)
{
    int first, i, i2;
    long less_than_cnt;

    if ( ! (*more) ) // assume first call, order of v is not set
    {
        // Start with the first permutation (ascending order).
        std::sort(v.begin(), v.end());
        *more = true;
        *even = true; // 1,2,3,4,... is always even (positive sign)
        return;
    }
    else // v has been ordered at least once before, continue on
    {
        std::next_permutation(v.begin(), v.end()); // this alters v
        // *more stays true

        // *even needs to be determined
        first = v[0];

        if ( (first % 2) == 1 ) { *even = false; }
        else { *even = true; }

        for (i = 1; i < n; i++)
        {
            less_than_cnt = 0;
            for (i2 = 0; i2 < i; i2++)
            {
                if (v[i2] < v[i]) { less_than_cnt++; }
            }

            less_than_cnt = less_than_cnt % 2;

            if (less_than_cnt == 0) // zero or even number of sign changes
            {
                if ( (v[i] % 2) == 1 ) { *even = !(*even); }
                // else even and anything = anything
            }
            else if (less_than_cnt == 1) // odd number of sign changes
            {
                if ( (v[i] % 2) == 0 ) { *even = !(*even); }
                // else even and anything = anything
            }
            else // this should never happen
            {
                PostError();
            }
        } // end for i
    } // end if *more was true
} // end NextPermute

```

## Appendix B – Source Code for Recursive Method

```
long CPolyMat::CalcDetRecurse(long start_index)
{
    long ret_val;
    long size, i, j;

    m_tmpPolyf.SetInteger(0);
    size = m_NumRows - start_index;

    ret_val = -1;    // default to failure

    if (start_index == 0)
    {
        m_RecurseDet.SetInteger(0);
    }

    // TODO: Should check that m_Mat[start_index][start_index] is NOT zero

    if (size < 1)        // shouldn't happen
    {
        m_RecurseDet = m_Mat[0][0];
    }
    else if (size == 1)
    {
        m_RecurseDet = m_Mat[m_NumRows-1][m_NumCols-1];
        ret_val = 1;
    }
    else
    {
        // Set up the submatrix - notice we alter m_Mat, destroying it BUT save memory
        for (i=start_index+1; i < m_NumRows; i++)
        {
            for (j=start_index+1; j < m_NumRows; j++)
            {
                m_tmpPolyf = m_Mat[i][start_index] * m_Mat[start_index][j];
                m_Mat[i][j] *= m_Mat[start_index][start_index];
                m_Mat[i][j] -= m_tmpPolyf;
            } // end for j
        } // end for i

        ret_val = CalcDetRecurse(start_index + 1);
        // m_tmpPolyf = result of the above call

        if (ret_val == 1) // success
        {
            for (i=0; i < size - 2; i++)
            {
                m_RecurseDet /= m_Mat[start_index][start_index];
            }
        }
    }

    return ret_val;
} // end DetRecurse
```

## Appendix C – Source Code for Gaussian Method

```
long CPolyMat::MakeUpper()
{
    CPolyfrac tmp_frac;
    long mat_size, last_row;
    long i, j, k;
    long num_swaps;
    bool zero_on_diag, swapped;
    long ret_val;

    ret_val = 1; // default to success
    num_swaps = 0;
    zero_on_diag = false;

    // For this to work for determinant calcs rows must = cols
    // this algorithm was based on that assumption (it might still work)
    if (m_NumRows != m_NumCols)
    {
        return 0;
    }

    mat_size = m_NumRows;
    last_row = mat_size;

    // Init tmp to all zeros - is done by the CPolyFrac constructor
    num_swaps = SetFirstRowForElim(); // guarantees m_Mat[0][0] is NOT zero
    // returns < 0 if NOT possible to do so

    if (num_swaps < 0) // unable to setup row[0]
    {
        return 0;
    }

    // row 0 stays unaltered
    for (i=1; i < last_row; i++) // last_row = mat_size unless get an all zero row
    {
        for (k=0; k < i; k++)
        {
            tmp_frac = m_Mat[i][k] / m_Mat[k][k];

            for (j = k+1; j < mat_size; j++)
            {
                m_Mat[i][j] = m_Mat[i][j] - (tmp_frac * m_Mat[k][j]);
            }
        }

        // It is now possible we set a diagonal element to zero
        // so we shall do column swaps to fix this
        // If this fails then row i is all zeros, so
        // there would be a zero on the diagonal somewhere anyway,
        // and we will leave it here.
        if (m_Mat[i][i].IsZero())
        {
            j = i+1; // all cols left of col[i] will have zeros in row[i]
            swapped = false;
            while ((!swapped) && (j < mat_size))
            {
                if ( !m_Mat[i][j].IsZero() )
                {
                    SwapCols(i, j);
                    swapped = true;
                    num_swaps++;
                }
                j++;
            }

            if (!swapped)
            {

```

```

        zero_on_diag = true;
        // could just return zero here
        // instead swap this row with current "last" row
        // and decrement number of rows to look at
        // (as the last ones will be all zero)
        SwapRows(i, last_row - 1);
        last_row--;
    }

} // end if [i][i] is zero
} // end for i

if (num_swaps % 2 == 0)
{
    ret_val = 1;
}
else
{
    ret_val = -1;
}

if (zero_on_diag)
{
    // determinant must be zero, and we moved zero rows to bottom of mat
    ret_val = 0;
}

return ret_val;
} // end MakeUpper

```