Performing Operations on Matrices using C

Joe Salkeld

30th October 2020

Abstract

The project aim was to write a program that would read in a matrix of rank n and perform various operations on the matrix, before writing out the matrix to a specified file. This was achieved by using the programming language "C" [1] and writing functions which can be called from the command line via simple strings. It was found that whilst the code could handle relatively small sizes of matrix quickly, the speed the code performed at slowed down dramatically as n increased. For $n \leq 9$ the code can perform the operations almost instantaneously, but for $n = 10$ the inverse calculation took a few seconds and for $n = 11$ it took around 30 seconds, and $n = 12$ took around 6 minutes. The result of this assignment suggests that alternative methods of matrix calculation need to be used when coding, as for matrices $n \geq 11$ the time taken to perform some operations became too slow for practical use.

1 Introduction

This project was a coding assignment in the C programming language. Its aim was to demonstrate the limits of coding when tackling linear algebra in a traditional sense, and therefore aims to show why alternative methods of calculation should be used when programming. The program written was to read in a randomly generated matrix of rank n as specified in a separate piece of code written by Dr C.D.H. Williams, and then perform various user-specified operations on the matrix, before outputting the answer in the same matrix format as it was read in. This allows for an outputted matrix to be reread in and processed again.

The functions to be performed on the matrix were: the Frobenius norm [2], the transpose [3], the matrix product [4], the determinant, the adjoint and the inverse [5]. Alongside this the functions all had checks in place to make sure that the calculation was possible, for example checking that if the determinant was requested the matrix input was a square matrix, of equal rows and columns.

2 Method

The basic structure of the code contains a main() function which has a set of case options performed using the getopt-long() function [6]. This allows the code to perform different functions based on the user input. Each of the cases then called a read function to read in the matrix specified by the user, followed by a function which took this matrix and performed the relevant calculation after making checks, before outputting an answer which was written to a file under the name specified by the user.

2.1 Reading in the Matrix

The first function written was to read in the matrix specified by the user. This was a void function which had two structures passed to it, which contained space for the file to be read in and stored, and then space for the useful information to be stored from the whole file. An error check was input first, which returns an error message if the filename that was input by the user cannot be found. This then quits the program instantly. Once the file has been opened, a while loop is written which runs through the entire file and counts the number of "words" in the file (where words are defined as any string of characters separated by a space) and then stores each "word" in a matrix of strings.

The program then loops through this matrix and does a comparison using strcmp() [7] on the strings. Firstly it looks for the word "matrix" and stores its location, and then it looks for the word "end", and stores its location. These two words define the beginning and end of the matrix, and so by finding these the data written in the header of the file is ignored.

The two values immediately after the word "matrix" are the number of rows and columns, and so these are read in and stored, as well as their value multiplied together, to get the entire size of the matrix.

Finally, the code loops through all the values after the row and column count but before the word "end" and stores each string in a new one-dimensional matrix, converting the string into a float in the process using atof() [8]. This matrix is one that has been defined in the structure initially passed to the function, and so its value is stored globally for future use.

A one-dimensional array was used because it is quicker to look up values and loop through them, so will help speed up the calculation for larger matrices.

2.2 Frobenius Norm

The Frobenius norm is defined as the square root of the sum of the absolute squares of its elements [2]. For a matrix $m \times n$, it equals

$$\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} \tag{1}$$

This is a very quick calculation to compute, as it requires a single loop which goes through each value in the matrix and squares it, before adding that value to a running total of squares. After the loop is complete the square root of that sum is taken and is returned by the function.

2.3 Transpose

The transpose of a matrix is defined as the matrix formed when the rows and columns of a matrix are swapped, so for a matrix A, size $m \times n$, the transpose is given as A^T, size $n \times m$ [3].

This is coded by a single loop, which goes through the values of the matrix, and stores each one in a new location in the output matrix.

Algorithm 1 Calculate the Transpose

```
Require: $i = 0$
  for $i = 0$; $i <$ rows; $i$++ do
Require:    $j = 0$
    for $j = 0$; $j <$ cols; $j$++ do
      transpose[rows*j +1] = intial[cols*i +j]
    end for
  end for
```

The two for loops are necessary to traverse the 2-D matrix stored in a 1-D array, and allow for rows and columns (i and j respectively) to be swapped over.

2.4 Matrix Product

The matrix product is the multiplication of two matrices, A, size $m \times n$ and B, size $n \times o$ done so the final matrix C has elements c_{ij} which are the sum of the multiplication of elements in row i of the A with those in column j of B, given as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{2}$$

where $i = 1, ..., m$ and $j = 1, ..., o$ [4].

When coding this function requires two input matrices to multiply together. This requires a workaround as the getoptlong() function only allows a single argument per case. To do this the case reads both the argument and the value of the string in the (argument+1) place and passes these values to the read function to be used, and then manually adds one to the argument so that the getoptlong() function moves beyond the second argument on to any further user input.

Once the two matrices are read in, the matrix product function loads these from the structure they are stored in, and first performs a simple check that the number of rows of matrix A are equal to the number of columns of matrix B. It then runs a loop as with the transpose, but adding a third loop which increments a running total for the value of each element c_{ij} until the multiplication is done where it is then stored in the output matrix before moving on to calculating the next element in C.

2.5 Determinant

The determinant for a 2×2 matrix is defined as

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc \tag{3}$$

and for a 3×3 matrix is defined as

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix} \tag{4}$$

where in this equation each 2×2 matrix is known as a minor of matrix A. For matrices of rank $n > 3$, they are split into their minor matrices and then calculated. For a 5×5 matrix, it is split into 4×4 matrices which are then split into 3×3 matrices which are then split into 2×2 matrices and then everything is calculated back up the initial 5×5 matrix [5].

To code this a recursive function is needed. This is achieved by calling the determinant function within itself. In order to prevent it from repeating forever, an if statement is made, stating that if the matrix is 2×2, then calculate and return the determinant using equation (3). If the matrix is not a 2×2 function, then a new array is created with dimensions of (rows-1) and (columns-1). It then stored values from the previous matrix into this new array, excluding any values that were in the ith row and jth column, in essence creating a minor matrix. A running total of the determinant is then incremented with the determinant function being called again inside this. In this way the function keeps creating minor matrices until it creates a 2×2 matrix, then uses this to calculate the matrix one step up, and so on until it has calculated the determinant for the initial matrix, at which point it returns that value.

A check is added into the code for a matrix that is 1×1, which returns the determinant being equal to that value.

2.6 Adjoint

The adjoint of a matrix is the transpose of the matrix of cofactors created. A matrix of cofactors is the matrix formed when calculating the value of the determinant of that positions minor matrix multiplied by the value in that position of the matrix [5]. For example a matrix A

$$A = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \tag{5}$$

would have a cofactors C_{ij} of

$$\begin{aligned} C_{11} &= a\begin{vmatrix} e & f \\ h & i \end{vmatrix} \\ C_{12} &= -b\begin{vmatrix} d & f \\ g & i \end{vmatrix} \\ C_{13} &= c\begin{vmatrix} d & e \\ g & h \end{vmatrix} \end{aligned} \tag{6}$$

and so on for the entire matrix. In order to get the adjoint of A the transpose of this will then need to be taken.

In the function, after checking for the squareness of the input matrix, it checks for a 1×1 matrix, as this is a special case which does not require any recursive calculation, it is defined as

$$adj(D) = 1 \tag{7}$$

where D is a 1×1 matrix, and so can be dealt with quickly.

For $n > 1$ however, a similar function to the determinant function is needed. This recursive function looks the same as the determinant function, except instead of having a running total for the determinant, it has an array that stores the values for each calculation. The determinant of the minor matrices is calculated by calling the previously explained determinant function recursively.

Once the cofactor matrix has been created, the transpose of it is taken using the same method as explained in section 2.3.

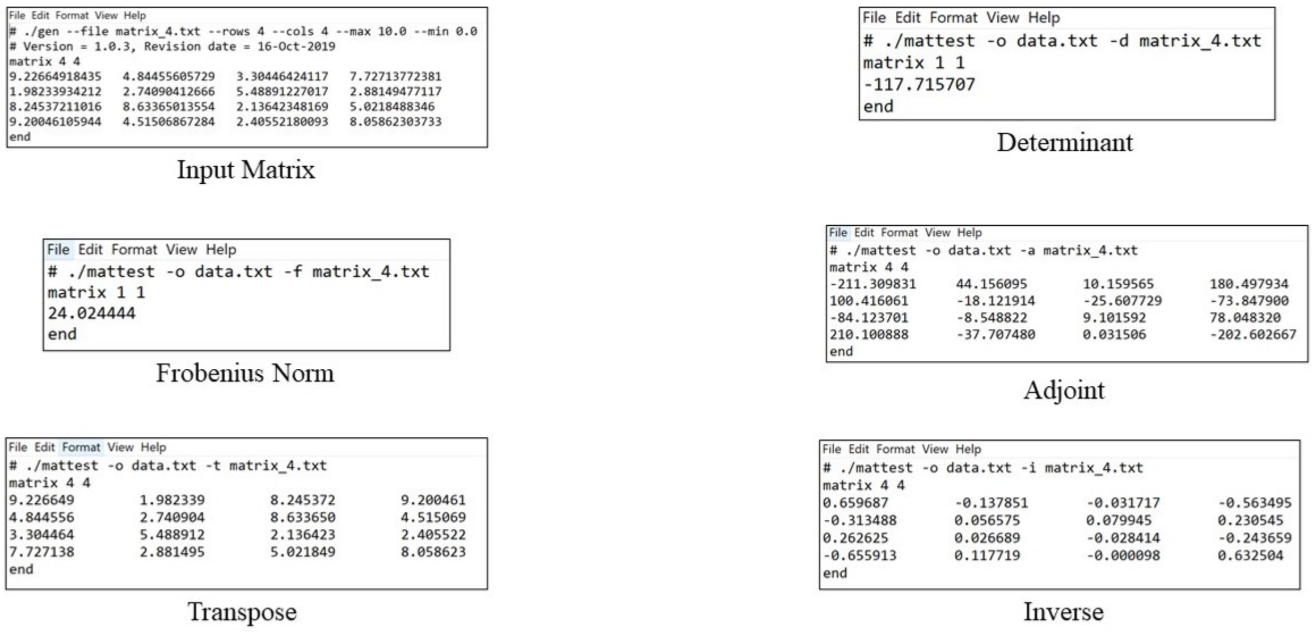
2.7 Inverse

The inverse of a matrix is calculated by multiplying the inverse of the determinant with the adjoint matrix [5]. It is denoted by A^{-1}, given formally as

$$A^{-1} = \frac{1}{|A|} adj(A). \tag{8}$$

For the special 1×1 case the code just takes a straight inverse of the value in the matrix and returns it. For $n > 1$ this requires the use of previous functions. Firstly the adjoint matrix is calculated in the same manner as previously described in section 2.6. Then the determinant function is called to calculate and return the value of the determinant. Finally a new output matrix loops through all the values of the adjoint, multiplies them by the inverse of the determinant and stores them.

3 Results



```
# ./gen --file matrix_4.txt --rows 4 --cols 4 --max 10.0 --min 0.0
# Version = 1.0.3, Revision date = 16-Oct-2019
matrix 4 4
9.22664918435   4.84455605729   3.30446424117   7.72713772381
1.98233934212   2.74090412666   5.48891227017   2.88149477117
8.24537211016   8.63365013554   2.13642348169   5.0218488346
9.20046105944   4.51506867284   2.40552180093   8.05862303733
end
```

Input Matrix

```
# ./mattest -o data.txt -d matrix_4.txt
matrix 1 1
-117.715707
end
```

Determinant

```
# ./mattest -o data.txt -f matrix_4.txt
matrix 1 1
24.024444
end
```

Frobenius Norm

```
# ./mattest -o data.txt -a matrix_4.txt
matrix 4 4
-211.309831     44.156095       10.159565       180.497934
100.416061      -18.121914      -25.607729      -73.847900
-84.123701      -8.548822       9.101592        78.048320
210.100888      -37.707480      0.031506        -202.602667
end
```

Adjoint

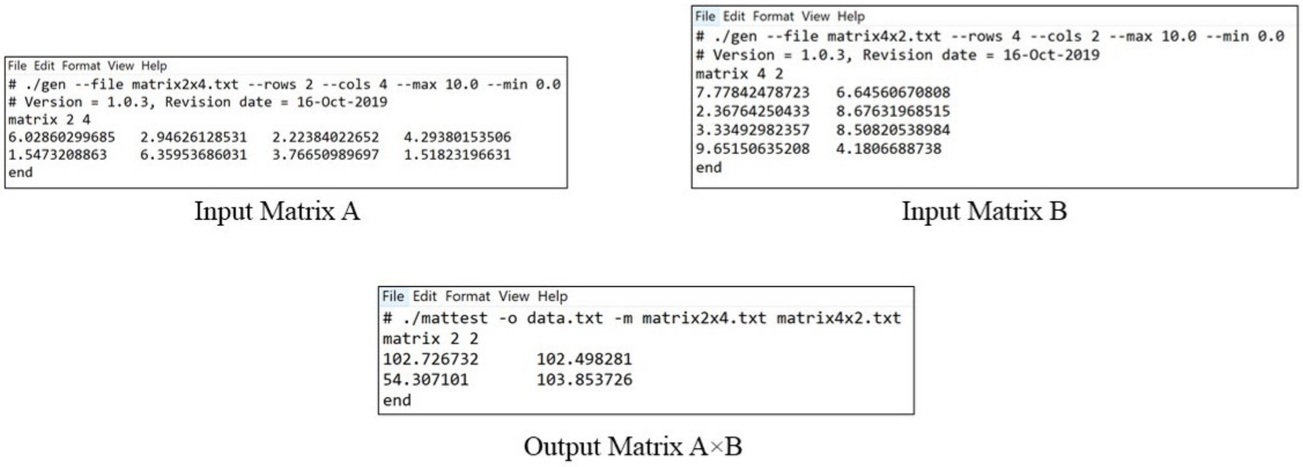
```
# ./mattest -o data.txt -t matrix_4.txt
matrix 4 4
9.226649        1.982339        8.245372        9.200461
4.844556        2.740904        8.633650        4.515069
3.304464        5.488912        2.136423        2.405522
7.727138        2.881495        5.021849        8.058623
end
```

Transpose

```
# ./mattest -o data.txt -i matrix_4.txt
matrix 4 4
0.659687        -0.137851       -0.031717       -0.563495
-0.313488       0.056575        0.079945        0.230545
0.262625        0.026689        -0.028414       -0.243659
-0.655913       0.117719        -0.000098       0.632504
end
```

Inverse

Figure 1: Input matrix and output files for varying functions



```
# ./gen --file matrix2x4.txt --rows 2 --cols 4 --max 10.0 --min 0.0
# Version = 1.0.3, Revision date = 16-Oct-2019
matrix 2 4
6.02860299685   2.94626128531   2.22384022652   4.29380153506
1.5473208863    6.35953686031   3.76650989697   1.51823196631
end
```

Input Matrix A

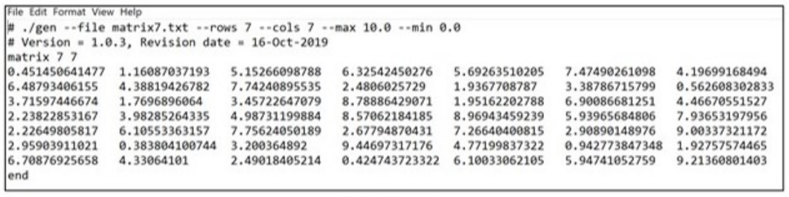
```
# ./gen --file matrix4x2.txt --rows 4 --cols 2 --max 10.0 --min 0.0
# Version = 1.0.3, Revision date = 16-Oct-2019
matrix 4 2
7.77842478723   6.64560670808
2.36764250433   8.67631968515
3.33492982357   8.50820538984
9.65150635208   4.1806688738
end
```

Input Matrix B

```
# ./mattest -o data.txt -m matrix2x4.txt matrix4x2.txt
matrix 2 2
102.726732      102.498281
54.307101       103.853726
end
```

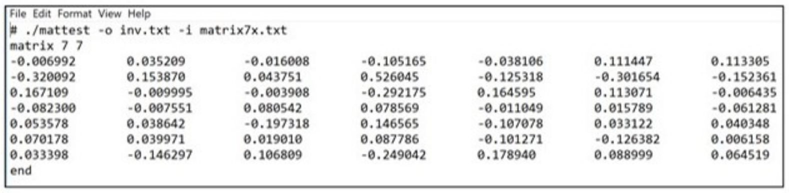
Output Matrix A×B

Figure 2: Result of a 4x2 multiplication with a 2x4



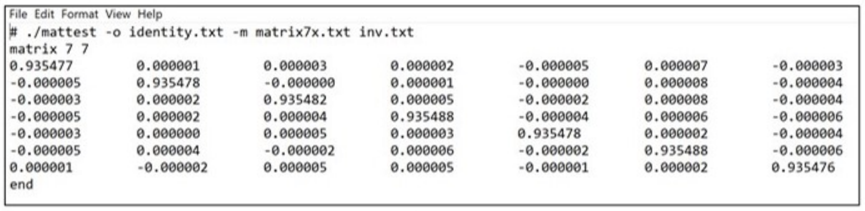
```
# ./gen --file matrix7.txt --rows 7 --cols 7 --max 10.0 --min 0.0
# Version = 1.0.3, Revision date = 16-Oct-2019
matrix 7 7
0.451450641477  1.16087037193   5.15266098788   6.32542450276   5.69263510205   7.47490261098   4.19699168494
6.48793406155   4.38819426782   7.74240895535   2.4806025729    1.9367708787    3.38786715799   0.562608302833
3.71597446674   1.7696896064    3.45722647079   8.78886429071   1.95162202788   6.90086681251   4.46670551527
2.23822853167   3.98285264335   4.98731199884   8.57062184185   8.96943459239   5.93965684806   7.93653197956
2.22649805817   6.10553363157   7.75624050189   2.67794870431   7.26640400815   2.90890148976   9.00337321172
2.95903911021   0.383804100744  3.200364892     9.44697317176   4.77199837322   0.942773847348  1.92757574465
6.70876925658   4.33064101      2.49018405214   0.424743723322  6.10033062105   5.94741052759   9.21360801403
end
```

Matrix A



```
# ./mattest -o inv.txt -i matrix7x.txt
matrix 7 7
-0.006992       0.035209        -0.016008       -0.105165       -0.038106       0.111447        0.113305
-0.320092       0.153870        0.043751        0.526045        -0.125318       -0.301654       -0.152361
0.167109        -0.009995       -0.003908       -0.292175       0.164595        0.113071        -0.006435
-0.082300       -0.007551       0.080542        0.078569        -0.011049       0.015789        -0.061281
0.053578        0.038642        -0.197318       0.146565        -0.107078       0.033122        0.040348
0.070178        0.039971        0.019010        0.087786        -0.101271       -0.126382       0.006158
0.033398        -0.146297       0.106809        -0.249042       0.178940        0.088999        0.064519
end
```

Inverse A^{-1}



```
# ./mattest -o identity.txt -m matrix7x.txt inv.txt
matrix 7 7
0.935477        0.000001        0.000003        0.000002        -0.000005       0.000007        -0.000003
-0.000005       0.935478        -0.000000       0.000001        -0.000000       0.000008        -0.000004
-0.000003       0.000002        0.935482        0.000005        -0.000002       0.000008        -0.000004
-0.000005       0.000002        0.000004        0.935488        -0.000004       0.000006        -0.000006
-0.000003       0.000000        0.000005        0.000003        0.935478        0.000002        -0.000004
-0.000005       0.000004        -0.000002       0.000006        -0.000002       0.935488        -0.000006
0.000001        -0.000002       0.000005        0.000005        -0.000001       0.000002        0.935476
end
```

Identity Matrix $A \times A^{-1}$

Figure 3: Result of a multiplication of a matrix with its inverse, which gives the Identity matrix of 1 on the leading diagonal and 0 everywhere else

Table 1: Table showing average processing time for the three time consuming functions for varying rank $n \geq 9$, run 10 times each

n	Determinant / s	Adjoint / s	Inverse / s
9	0.021 ± 0.007	0.189 ± 0.002	0.211 ± 0.008
10	0.215 ± 0.007	2.092 ± 0.005	2.307 ± 0.010
11	2.320 ± 0.016	25.573 ± 0.097	27.791 ± 0.088
12	27.743 ± 0.067	334.173 ± 1.490	359.174 ± 0.674

Timings were attempted to be made for the Frobenius norm, the transpose and the matrix product, but the code performed all these calculations up to a 12×12 matrix in less than a millisecond which was the precision of the timing clock, and so no useful data was recorded. Similarly, for matrices of $n \leq 8$, timings were again too small to be resolved for the determinant, adjoint and inverse functions.

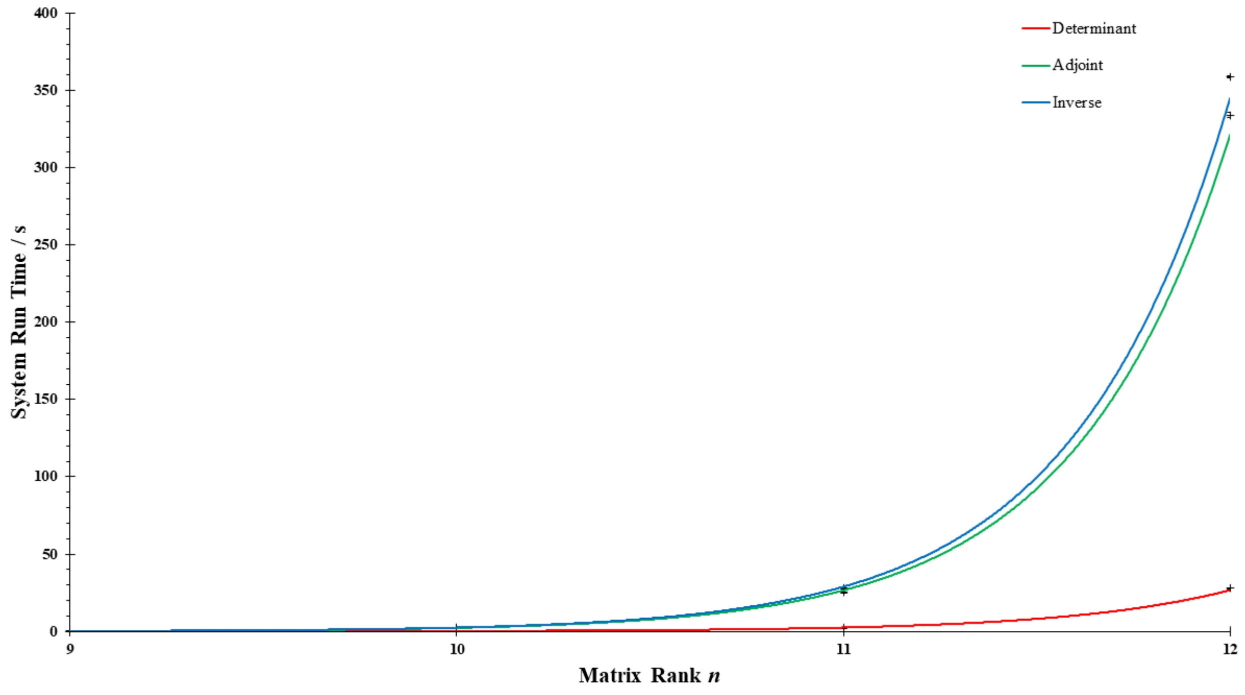


Figure 4: Time taken for the determinant, adjoint and inverse to be calculated for varying rank n matrices

4 Discussion

The code written has been shown to perform the required functions up to matrices of the order 12×12. However, there are some noticeable errors in some of the results, as well as a sharp increase in run time for matrices above $n = 9$, which can be seen in Figure (4).

Looking at Figures (1) and (2), there are a few things to notice. The accuracy of the data has decreased from 11 decimal places down to 6 decimal places, and values have been truncated at the sixth decimal point, although they have been rounded to that point. The easiest place to notice this is looking at element [4,2] in the input matrix and then element [2,4] in the transpose matrix. Here the sixth decimal point has been rounded up from an 8 to a 9 as a result of the following digit. This truncation of precision is due to the conversion from the input numbers being stored as a string to being stored as a double using the atof() function [8]. This function turns floats into doubles but only allows for 6 decimal places. In order to achieve more accurate precision, a different method of reading in the data would be necessary. One possible solution would be to read values character by character instead of by groups of characters separated by a space. This would allow for the full number to be read and stored as a double. This truncation is responsible for the precision of all calculated values to only have 6 decimal places. All functions and operations are being done using the double type, and so were the values stored by the code to have all 11 decimal places of precision, all the operations would compute the answer accurate to that many places. In this way the improvement to the code would simply require a method of storing the read in variables to a greater precision, and all the functions would need no editing. Outside of this precision however the functions perform accurately. It is expected that the run times would increase however for values with all 11 decimal places, which would likely make 12×12 matrices too large to run effectively.

Figure (3) showcases a property that matrices have, namely that a matrix multiplied by its inverse equals an identity matrix, a matrix of rank n containing zero except for on the leading diagonal which contains one. From this figure, the output of a 7×7 matrix gives a very rough approximation of an identity matrix. This is most likely because of the shortening of values for the matrix, although the values for the leading diagonal seem to be consistently far off from 1. On average the zero values are equal to 4×10^{-7}, which is within the error for six decimal places. However the average for the leading diagonal is 0.935481, which is around 6.5% smaller than it should be. This error seems too large to have come from truncation of the values, and so further investigation would be needed into why this margin of error is the size it is.

The speed of calculation for the Frobenius norm, the transpose and the product were all too quick to be timed for every n tested, up to 12. This is to be expected, as all three of these perform simple linear calculations which just require looping through every value and performing a single action to it. By storing the matrix as a 1-D array, the loop can just increment a single value until it reaches the end of the array. A comparison between these run times and ones where the matrix is stored as a 2-D array would be needed to determine whether this makes a noticeable difference, as well as performing the calculation for much larger values of n to allow for some time to be recorded. Timing values could potentially be calculated if a more accurate way of timing the code could be used. The inbuilt time() function in Cygwin [9] which was used to compile and run the code only shows the time to the nearest millisecond, which works well for timing larger matrices, but an inbuilt function could be coded into the C program that timed the functions more accurately to allow for more comparison.

The data for the determinant, adjoint and inverse also follow expected run times, with all matrices of $n \leq 8$ having run times shorter than the resolution of the timing clock. The expected time for running the code is that the adjoint will take longer than the determinant, and the inverse will take longer than the adjoint, and that can be clearly seen in Figure (4). This is because in order to calculate the adjoint or inverse, a matrix of cofactors is created, which requires the same process as the determinant function but applied to all elements not just the first row. The inverse would be expected to take longer than the adjoint as it also requires calculation of the full determinant of the matrix and then multiplication on all the elements. As a result of this calculation, the adjoint and inverse functions would be predicted to take around n times longer than the determinant, where n is the dimension of the matrix. From the data in Table (1), it can be seen that this trend is followed, for $n = 10$, the run time for the adjoint is 9.73 times longer than for the determinant, and this trend is followed upwards, where for $n = 11$, the run time for the adjoint is 11.02 times longer, and for $n = 12$ it is 12.05 times longer.

The run time data is taken from an average of running the program ten times for each function on different random matrices of the same rank n. The errors were then calculated from this. Looking at figure (4) it can be seen that the run time increases exponentially with n, meaning that the program, whilst working efficiently for small n, quickly becomes unusable. If the computing time continued on this trend, by $n = 13$ the run time would be over an hour for the inverse, and for $n = 14$ it would be around a month. All the values for data in this report were taken from matrices with elements between 0-10, so a different analysis could be done on whether the minimum and maximum values of the elements in the matrix affects the run time also.

When checking for the accuracy of the matrix calculations using a simple Microsoft Excel spreadsheet, the results were returned in less than a second for all matrices, and for all functions required. This suggests that following known linear algebra techniques for matrix calculation is not the most efficient way for a program to do matrix calculation, and so a further investigation into alternative methods of calculation and their speeds would be necessary, as currently the usefulness of the program written ceases at around $n = 11$, as values above that are either too large to be processed at all or take too long when using recursive functions.

5 Conclusion

The aim of the assignment was to write a program in C that would read in and store a randomly generated matrix of size n and then perform various operations on that matrix whilst also performing some simple checks, for example if the determinant function was needed that the matrix read in was square. The program also had to output a file of the same type as was input, to allow it to be reread and have more operations performed on it.

It was found that the code could perform the Frobenius norm, transpose and product functions practically instantaneously, and that for matrices smaller than $n = 9$, could also perform the determinant, adjoint and inverse functions in less than a millisecond. For increasing n, the time taken to calculate the determinant, adjoint and inverse grew drastically, with timing of the order of a few seconds for $n = 10$ to of the order of 5 minutes for $n = 12$.

On analysis of the code, a further investigation would be needed into alternative methods of matrix calculation, as the traditional methods used in this assignment proved that they handle smaller matrices well, but slow down very quickly as n increases.

References

[1] C Programming Language reference library. https://en.cppreference.com/w/c/language. (Accessed 29/10/20).

[2] Gene Howard Golub and Van, C.F. (1996). Matrix computations. Baltimore, Md.: Johns Hopkins University Press.

[3] Whitelaw, T.A. (1999), p39. Introduction to Linear Algebra. Boca Raton: Chapman & Hall/Crc.

[4] Whitelaw, T.A. (1999), p31. Introduction to Linear Algebra. Boca Raton: Chapman & Hall/Crc.

[5] Khamsi, M. A. Determinant and inverse of matrices. http://www.sosmath.com/matrix/inverse/inverse.html (1999). (Accessed 29/10/20).

[6] GetOptLong() function description. https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html. (Accessed 29/10/20).

[7] Strcmp() function description. https://www.programiz.com/c-programming/library-function/string.h/strcmp#:~:text=The%20strcmp()%20compares%20two,is%20defined%20in%20the%20string.. (Accessed 29/10/20).

[8] Atof() function description. https://www.tutorialspoint.com/c_standard_library/c_function_atof.htm. (Accessed 29/10/20).

[9] Cygwin C Compiler. https://cygwin.com/install.html. (Accessed 29/10/20).