# PHYM004 Homework 1

Tom Joshi-Cale

27th October 2020

**Abstract**

This document aims to give a competent programmer the ability to understand and reproduce the code found in mat_test.c, a code created to perform linear algebra calculations on inputted matrices. It also contains an analysis of execution speed and precision of that code, as well as an evaluation of its results, and the methods used to obtain them. It is concluded that the cofactor expansion method of calculating a determinant is too computationally heavy to be of use for matrices of large dimensions, and shortcuts like Gaussian Elimination should be employed.

## 1 Introduction

Many physical, scientific and computer science problems can be modelled and solved as a system of linear equations which can be solved with linear algebra. For example, an early version of Google's page-rank algorithm employed linear algebra [1], and linear algebra is also used in mechanics to solve the equations of motion for N bodies [2]. For systems with large N, it becomes necessary to solve the problems computationally, and they can be solved quickly and accurately if the correct methods are used.

To investigate a method of solution to linear algebra using direct matrix calculations, a program named mat_test.c was written in C, which needed able to read in either one or two matrices that are produced by the mat_gen.c code [3], and perform any of 6 tasks on request. The code had to be able to calculate the Frobenius Norm (sometimes known as the Euclidean norm) and the transpose of any inputted matrix; the determinant, adjoint and inverse of any inputted square matrix; and the matrix product of any two compatible inputted matrices.

The code had to be well-structured, with a summary comment at the top of the file, have appropriate levels of modularisation (i.e. correctly use functions to only run lines of code that are needed)and it also had to be able to recognise an incorrect or invalid input and appropriately warn the user of the errors.

This exercise was also intended to "demonstrate how poorly some straightforward methods can perform" [4], and an investigation into this can be found in the Conclusion section of the report.

## 2 Background Theory

As the six necessary tasks all involve matrix calculations, there are various techniques that the code employs. The Frobenius Norm, or Euclidean Norm, of an $M \times N$ matrix $\mathbf{A}$ is:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{M} \sum_{j=0}^{N} |a_{ij}|^2} \tag{1}$$

where $|a_{ij}|$ is the matrix element on the $i$th row in the $j$th column of the matrix $\mathbf{A}$.

The second task, finding the transpose of the matrix, is another relatively straightforward task. To calculate the transpose of a matrix $\mathbf{A}^{\mathrm{T}}$, each element of the original matrix $\mathbf{A}$ has its row and column position swapped:

$$a_{ij}^{\mathrm{T}} = a_{ji}. \tag{2}$$

Next, the matrix product of two inputted matrices had to be calculated. For an $M \times N$ dimensional matrix $\mathbf{A}$, and an $N \times P$ dimensional matrix $\mathbf{B}$, the matrix product $\mathbf{C}$ is an $M \times P$ dimensional matrix, calculated as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \tag{3}$$

for $i = 1, ..., m$ and $j = 1, ..., p$. This means that in order to calculate the matrix product of two matrices, it first must be checked that the number of columns of the first matrix is equal to the number of rows of the second matrix. If they are not equal, then the calculation cannot be performed. Additionally, as matrix multiplication does not commute, the order of matrices is important, since $AB \neq BA$.

The determinant of an inputted matrix also had to be calculated. This is the most complicated step of the process, and the method used is that laid out by M. A. Khamsi [5]; for an N-dimensional square matrix $\mathbf{A}$ with elements $a_{ij}$, each element has a cofactor $A_{ij}$ which is defined as the determinant of the (N-1)-dimensional square matrix obtained by removing row $i$ and column $j$, and multiplied by $(-1)^{i+j}$; the determinant is:

$$\det(\mathbf{A}) = \sum_{j=1}^{j=N} a_{ij} A_{ij}, \tag{4}$$

for any constant $i$, or:

$$\det(\mathbf{A}) = \sum_{i=1}^{i=N} a_{ij} A_{ij}, \tag{5}$$

for any constant $j$.

Notice that the calculation of the determinant of an N-dimensional matrix requires the calculation of an (N-1)-dimensional matrix, so a function for the determinant that calls itself recursively is needed. The determinant of a 2-dimensional matrix of the form: $\mathbf{M} = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ can be calculated as:

$$\det(\mathbf{M}) = ad - bc, \tag{6}$$

which is a relatively simple calculation, so the determinant of an N-dimensional matrix needs to be calculated recursively until a 2x2 matrix is reached. In the event that the user is asking for a determinant of a 1x1 dimensional matrix, the answer is straightforward since the determinant of a 1x1 dimensional matrix is simply the value of the single element in the matrix, so no calculation is necessary.

The fifth task was to calculate the adjoint of an inputted square matrix. M.A. Khamsi defines the adjoint, denoted as adj($\mathbf{A}$) as the transpose of the matrix of cofactors. As a reminder, the cofactor of an element of matrix $\mathbf{A}$ ($a_{ij}$) is the determinant of the matrix left over when row $i$ and column $j$ are removed, and then the determinant multiplied by $(-1)^{i+j}$.

The final task was to calculate the inverse of the inputted square matrix. The inverse, denoted $\mathbf{A}^{-1}$ is calculated as:

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})}\text{adj}(\mathbf{A}), \tag{7}$$

which is a relatively straightforward calculation to implement once the previous tasks have been completed. It is possible to check the accuracy of a calculated inverse using the definition of an inverse of a matrix as a matrix which, when multiplied with the original matrix, results in the identity matrix, or, mathematically:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{8}$$

# 3 Method of Solution

## 3.1 Parsing the Command Line

The task chosen to be performed is read in from the command line using the getopt.h package, and a switch case is used to determine which functions to execute with which argument, as laid out in the getopt_long documentation [6].

Command line arguments are stored in the char array `argv`, and the number of arguments given is stored as an integer in `argc`. This fact is used to print the header of each output, which always consists of two lines of comments. The first comment is simply what command was used to generate the file, which is generated using a for loop from $i = 0$ to $i <$ `argc`, which prints `argv[i]`. This will always write whatever commands were used to execute the program. The second line just contains the Version Number and latest Revision Date, which are stored as manually updated global variables on lines 6 and 7 of mat_test.c

## 3.2 Reading in the matrices

Regardless of which task is chosen, it is always required to read in one inputted matrix and save it into an array. Since in C it is not possible to declare a variable sized array, it is necessary to know the dimensions of the inputted matrix before the variable containing it is declared. For this reason, first a function is called which parses the inputted text file, searching for a line

beginning with the letter 'm'. This is done because it allows for that line to be at any point of the text file as long as it follows the format 'matrix 3 2'.

Once the dimensions are known, another function is called which reopens the text file and reads through it line by line. If the line begins with an 'e', it is assumed to be the 'end' line, and exits the function; if the line begins with a '#' or an 'm', the line is assumed to be a comment line or the dimension declaration line and is skipped to the next line. All other lines are assumed to be part of the matrix, and read into a 2D array. Memory is then allocated for a 1D array using

```
double *array = calloc(rows * cols, sizeof(double));
```

and the 2D matrix is then mapped onto the 1D array as

```
array[cols*i+j] = matrix[i][j];
```

which is a technique for storing 2D arrays in contiguous memory for faster access [7]. At this point, the code branches depending on what task the user has requested, however in all cases the code confirms that the memory has been correctly allocated, before calling any relevant functions required to perform the task. Allocated memory is then freed, before the output is printed to stdout.

## 3.3 Frobenius Norm

For this task there is only one function that is called, which allocates a 1x1 matrix to store the norm, and then has a double for loop over each element in the matrix, squaring it and adding it to the running total for each elements. The total is then square rooted and returned back to main, where it is passed to the print_out function (Section 3.9).

## 3.4 Transpose

When finding the transpose of the inputted matrix, there is again only one function called; which allocates memory to a matrix of equal size to the input, then puts element $i, j$ from the inputted matrix into position $j, i$ into the newly allocated matrix via a double for loop, which is then returned back to main. The inputted matrix's memory is freed, and the new matrix is passed to print_out.

## 3.5 Matrix Product

This task is the only task which needs to read in a second matrix, so the first thing this case does is extract the dimensions and data for a second matrix. If both matrices have been correctly allocated, then they are passed to a function which calculates the matrix product. This function first checks whether the number of columns in the first matrix is equal to the number of rows in the second. If they are not equal then the code warns the user and exits, as it the inputted matrices are not compatible for matrix multiplication. If the two matrices are compatible, then a third matrix with dimensions COLS1 $\times$ ROWS2 has memory allocated for it, and then, using a

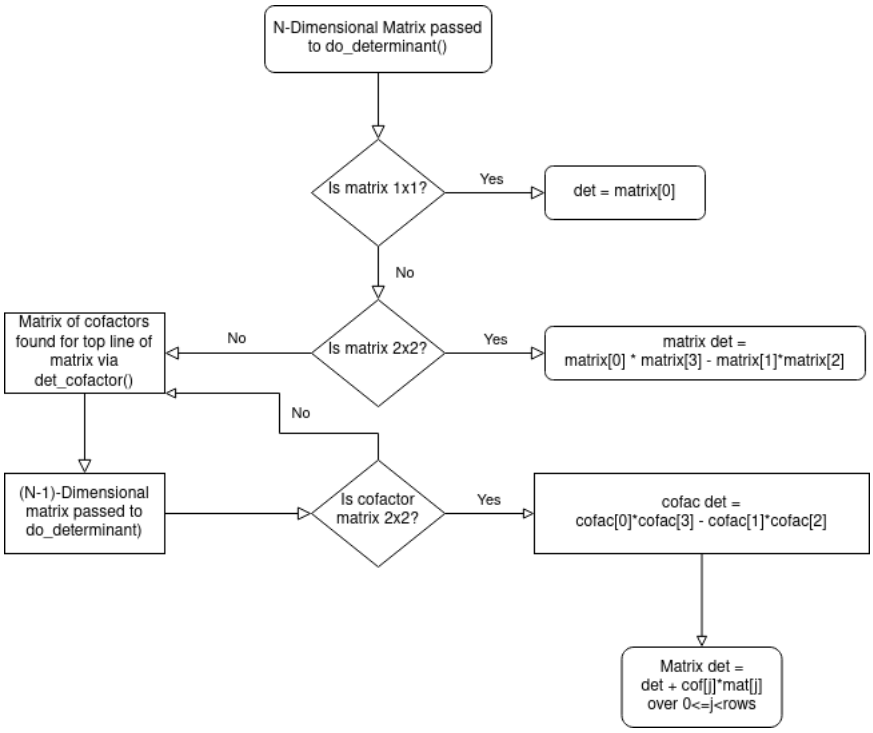


Figure 1: A flowchart showing how the do_determinant() and det_cofactor() functions recursively call each-other until the final determinant is calculated.

triple for loop, performs the calculation given in Equation (3). The new matrix is then returned out of the function, and the necessary output is printed to stdout.

## 3.6 Determinant

The determinant task, as well as both the following tasks (Adjoint and Inverse), must first check that the inputted matrix is square: i.e. they have an equal number of rows and columns. If they are not, then the user is warned and the code exits. If the matrix is a square matrix, then a function is called to perform the determinant. This function first allocates memory for a matrix of co-factors of dimensions 1xCOLS, then allocates memory for a 1x1 matrix to store the determinant in. The code then checks if the dimensions of the inputted square matrix are 1x1, in which case the determinant is simply the single element in the matrix. Next, the code checks whether the inputted matrix is 2x2. If so, it calculates the determinant as shown in Equation (6). Otherwise, i.e. if the dimensions are 3x3 or greater, the code passes to another function which calculates the 1XCOL dimensional matrix of cofactors for the top row of the matrix, and then calculates the determinant as shown in Equation (5), with the fixed $i$ being 0 (i.e. for the top row of the matrix). This process is shown as a flowchart in Figure 1. Finally, the code frees the memory the 1xCOL dimensional matrix of cofactors was stored in, and returns the determinant to main, where it is passed to the print_out function, after the memory used by the input matrix has been freed.

## 3.7 Adjoint

Like its predecessor, the case for which the adjoint must be calculated first checks whether the matrix is square, outputting an error message and exiting if not. If it is a square matrix, then a function is called to find the adjoint of the inputted matrix. This function allocates memory for a matrix, then checks to see whether the inputted matrix is a 1x1 matrix. If so, it sets the adjoint equal to 1, as for the trivial case of a 1x1 matrix the adjoint is the identity matrix by definition. If not, then the function fills the input matrix with a matrix of cofactors by calling a function to calculate the matrix of cofactors. This cofactor function works very similarly to the cofactor function called when calculating the determinant, but instead of only calculating the cofactors for the top row of the inputted matrix, it calculates a cofactor for every row of the matrix. This then passes back the matrix of cofactors to the adjoint function, which passes it to the transpose function explained in Section 3.4. This transposed matrix of cofactors is the adjoint, which is returned back to main and passed to the print_out function.

## 3.8 Inverse

The final task's case again begins by confirming that the inputted matrix is square in the same way as the adjoint and the determinant cases. If it is square, then the determinant function is called on the inputted matrix (Section 3.6). This case then checks if the matrix is singular (has a determinant of 0). If it is singular, then the code informs the user of its singularity, and finishes, as singular matrices have no inverse. If it is not singular (det! = 0), then the function allocates memory for an adjoint matrix and uses the adjoint function from the previous case (Section 3.7) to populate it with the adjoint of the inputted non-singular matrix, and then uses a double for loop to multiply each element of the adjoint matrix by $\frac{1}{\det(\mathbf{A})}$, where $\mathbf{A}$ is the inputted matrix. The result of this calculation is returned back to main, where it is finally passed to the print_out function.

## 3.9 Print Out

The final function within the code is again one that is called by every case, and simply prints the array in the same format as to match with the output of mat_gen.c. Compatability with this function is the reason why all the outputs of previous functions are formatted as arrays, even those which are just one number such as the Frobenius norm and the determinant; as it allows this function to handle both single value answers (as a 1x1 array) and multi-dimensional array answers. The function recieves a pointer to the required output, and the dimensions of the output array. It then prints 'matrix ROWS COLS', where ROWS and COLS correspond to the matrix's dimensions, and then uses a double for loop to print every value in the matrix. The precision chosen for this print is 12 decimal places, so as to match with the output of the mat_gen.c code. Finally, the function prints the word 'end'. This function uses a simple printf function to handle

the printing to stdout, whereas mat_gen.c uses fprintf to print to a text file. The reason for this difference is that this code is more likely than mat_gen.c to be used simply to see the results of a calculation without the need to save it, however the output is formatted in such a way that if the code is piped into a text file (for example by being run like `$ ./mat_test -d matrix1.txt > output.txt`), the generated output file can still be read in by the code itself.

# 4 Results

Each task was tested numerically against either calculations done by hand, examples found performed by other people, or against online matrix calculators. For example, the Frobenius norm was calculated by hand for multiple matrices, and compared to the output of mat_test.c. For example, inputting the file 2mat.txt, which contains the matrix

$$\begin{pmatrix} 0.813291693019 & 0.18341637551 \\ 0.547310722781 & 0.409208229002 \end{pmatrix}.$$

Evaluating the Frobenius norm numerically on a CASIO fx-991ES PLUS gives the value 1.078000625; and the Casio.com matrix norm calculator [8] calculates the Frobenius norm of that matrix as 1.0780006246255. The output of mat_test.c gives the Frobenius norm as 1.07800062463, which is a greater level of precision than that outputted by the Casio calculator. While this is less precision than the online Casio norm calculator, if the print_out functions precision is changed to a greater number of decimal places, then a higher level of precision can be obtained.

The accuracy of the Transpose matrix can be seen easily, as the numerical values within the matrix do not change value, only position within the matrix. Running it again on 2mat.txt, the output of mat_gen.c is

$$\begin{pmatrix} 0.813291693019 & 0.547310722781 \\ 0.183416375510 & 0.409208229002 \end{pmatrix}$$

from which it can easily be seen the transpose matrix has been correctly calculated.

To test the matrix product, a few more simple matrices were constructed manually. **Prod1** is

$$\begin{pmatrix} 2 & 5 & 7 \\ 3 & 2 & 4 \end{pmatrix}$$

and **Prod2** is

$$\begin{pmatrix} 1 & 3 \\ 8 & 5 \\ 2 & 4 \end{pmatrix}$$

.

Calculating the matrix product of **Prod1 Prod2** gives us the result

$$\begin{pmatrix} 56 & 59 \\ 27 & 35 \end{pmatrix}$$

which is verified by both calculation on a CASIO fx-991ES PLUS calculator, and the matrix.reshish.com calculator [9] as being correct. The output of the determinant function was also verified through both the matrix.reshish.com calculator, and the CASIO fx-991ES PLUS calculator.

The determinant and adjoint calculations were also verified against the numerical solutions given by M. A. Khamsi in their description of the method of calculating the determinant, adjoint and inverse, outlined in Section 3. Here the matrix $\mathbf{A} = \begin{pmatrix} 1 & 3 & 2 \\ -1 & 0 & 2 \\ 3 & 1 & -1 \end{pmatrix}$ is used, and the determinant is found to be 11, the adjoint is found to be $\begin{pmatrix} -2 & 5 & 6 \\ 5 & -7 & -4 \\ -1 & 8 & 3 \end{pmatrix}$. The mat_test.c code correctly calculates the determinant as 11, and also correctly outputs the adjoint matrix as expected above. The outcomes of this code can be seen within the mat_test.c submission folder to check this verification, as detA.txt and adjA.txt.

Finally, the accuracy of the inverse matrix calculation can be verified by multiplying the original matrix by the inverse matrix, and comparing the result with the identity matrix. Since the definition of the inverse matrix is that it should give the identity matrix when multiplied with the original matrix, this is an easy way to check its validity. When this is done with mat_test.c, the elements that are supposed to be zero are found to be exponential numbers on the order of $10^{-11}$ to $10^{-14}$, and the elements that should be one are often found to be one when rounded to 10 or 11 decimal places. All the square matrices tested, their inverses, and the result of the validity check can be found in the submission folder as Nmat.txt, Ninv.txt and Ninvcheck.txt respectively, where N represents the NxN dimensional square matrix.

# 5 Discussion

The most processor heavy, and therefore time-consuming, part of the code, is calculating the matrix of cofactors. Whilst each individual calculation is not too taxing, the sheer number of calculations that need to be performed when calculating the adjoint or inverse matrix means that as N increases, so does the time. At a previous point in the development of mat_test.c, it was not realised that the determinant only required the matrix of cofactors for the top row of the inputted matrix, so every time the determinant was calculated, each element of the matrix had its cofactor calculated, and each of these calculations required the determinant of an (N-1)x(N-1) matrix and so on until a 2x2 matrix is reached. This means that for an 8x8 matrix, for example, with 64 elements, each cofactor required the determinant of a 7x7 matrix, with 49 elements, which each required a determinant of a 6x6 matrix with 36 elements and so on,

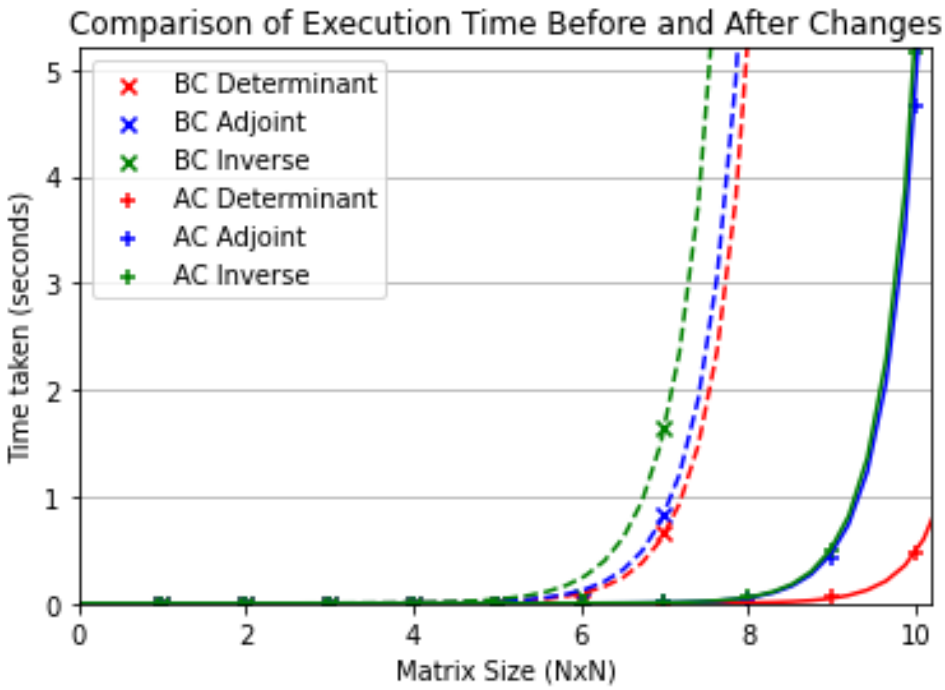


Figure 2: Plots of the execution time of the determinant, adjoint and inverse tasks both before optimisations were made to the code (BC) and after the changes were made (AC). Additionally, all trends are fitted with an factorial curve fit to predict how long the tasks would take for larger matrices.

leaving the total number of calculations required to find the determinant of an 8x8 matrix at $64 \times 49 \times 36 \times 25 \times 16 \times 9 \times 4$, or $1.63 \times 10^9$ calculations, following an $(N^2)!$ trend. When this error was spotted, and the code rewritten to only require the cofactors of the top row to be computed, and then each determinant for those only required the following top row, the number of calculations dropped to $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2$, or a total number of 40,320; which will still a large number is not as complex a calculation as before, following an $N!$ trend. The number still rises quickly, and Figure 2 compares the execution time of the determinant, adjoint and inverse calculations before and after this change was made to the code.

It can be seen that even optimisation are made to the running of the code, the execution time of the adjoint and inverse tasks that require the full cofactor matrix to be calculated, rises steeply as Matrix Size increases. In fact, for the inverse task where the time measured for a 10x10 matrix was 5.154 seconds, the factorial fit to data predicts that an 11x11 matrix would take 56.677 seconds and a 12x12 would take 680 seconds, or over 11 minutes to complete. Indeed, whilst the determinant is rising less steeply than the adjoint and inverse trends - taking only 0.473 seconds to calculate for a 10x10 matrix - curve fitting suggests that a 12x12 matrix would take 63 seconds, and a 13x13 over 813 seconds, or 13.5 minutes.

# Conclusions

Even a brief examination of the execution speeds of mat_test.c show clearly that this method of calculating the determinant; adjoint and inverse are too computationally heavy to be sensible choices for working with linear algebra of matrices larger than around 10x10, and the number of operations required increases factorially through cofactor expansion. When matrices reach this scale, it would be sensible to look for shortcuts to complete the work. One such shortcut is Gaussian Elimination [10], which uses three facts about the determinant to simplify the problem by converting the inputted matrix into an upper-triangular matrix (row-echelon form), at which point the determinant is just the product of the diagonal elements. By contrast to the factorial expansion in number of operations seen by cofactor expansion, the number of operations required for Gaussian expansion increases with the cube of $n$, so except for very small $n$, Gaussian expansion is a much more efficient and effective way of calculating the determinant and inverse of N-dimensional square matrices.

# References


[1] R. E. Tanase and R. A. Radu, "Pagerank algorithm - the mathematics of google search," lecture, Cornell University, 2009. http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html, accessed 29th October 2020.

[2] A. Pál and A. Süli, "Solving linearized equations of the n-body problem using the lie-integration method," *Monthly Notices of the Royal Astronomical Society*, vol. 381, pp. 1515–1526, 10 2007.

[3] C. D. H. Williams, *mat_gen.c*. https://vle.exeter.ac.uk/pluginfile.php/2108209/mod_resource/content/7/mat_gen.c, accessed 13 October 2020.

[4] C. D. H. Williams, *The Performance of Matrix Calculations*. https://vle.exeter.ac.uk/pluginfile.php/2108208/mod_resource/content/13/CW191027-01%20PHYM004-HW1.pdf, accessed 13 October 2020.

[5] M. A. Khamsi, *Determinant and inverse of matrices*. http://www.sosmath.com/matrix/inverse/inverse.html (1999), accessed 13 October 2020.

[6] B. Gough, *GNU scientific library reference manual*, ch. 25.2.4. Network Theory Ltd., 2009.

[7] C. D. H. Williams, *PHYM004 P04a Arrays*. https://vle.exeter.ac.uk/pluginfile.php/2193821/mod_resource/content/1/P-04a%20Arrays.pdf Accessed 13th October 2020.

[8] *Matrix Norm Calculator*. https://keisan.casio.com/exec/system/15052019544540, accessed 29th October 2020.

[9] *Matrix Multiplication Calculator*. https://matrix.reshish.com/multCalculation.php, accessed 29th October 2020.

[10] P. Young, "Solving linear equations by gaussian elimination," lecture handout, University of California Santa Cruz, 2014. https://young.physics.ucsc.edu/116A/gauss_elim.pdf, accessed 29 October 2020.