Computational Physics 301: Exercise 1 - Matrices and Linear Algebra

Caspar Fuller - 1310006 *15 February 2015*

Matrix Inversion

Linear algebra is a central component of mathematics and linear systems can often be presented in matrix-vector form as shown in equation *(1)*, where A is the matrix of coefficients, x is the unknown vector of variables and b is the known vector of constants. Properties of matrices can then be exploited to provide quick and efficient ways to solve linear systems. This report aims to implement and analyse 3 simple matrix techniques for solving linear systems, investigating the stability and efficiency of each one. Specifically I shall be using these techniques to find the inverse of a given matrix. These techniques will then be applied to a real world physics problem. Each program is written in the C language.

$$Ax = b \tag{1}$$

Problem 1: Recursive Determinants

Theory

This first task will use the standard inversion formula, equation *(8)*, to calculate the inverse of an arbitrary *nxn* square matrix. The inverse of a matrix is crucial for the solution of linear systems as by pre-multiplying both sides of equation *(1)* by the inverse of A (equation *(2)*), a solution for x can be found, shown in equation *(3)*.

$$A^{-1}Ax = A^{-1}b \tag{2}$$

$$x = A^{-1}b \tag{3}$$

The inverse of A, is given by equation *(4)*, where $|A|$ is the determinant of A and *adj.A* is the adjoint of A. The adjoint is the transpose of the matrix of co-factors for A, given as the symbol C^T.

$$A^{-1} = \frac{adj.A}{|A|} \tag{4}$$

Methods to find the determinant and matrix of co-factors for an arbitrary matrix use the powerful technique of recursion, the repeated application of an entire procedure within that same procedure.

To calculate the determinant of an arbitrary matrix the Laplace expansion[1] can be used. This expansion, shown in equation *(5)*, can be applied to a general *nxn* matrix by expressing the determinant as determinants of smaller sub-matrices of size *n-1 x n-1*. The Laplace expansion occurs along a specific row or column where each matrix element on the row is multiplied by the corresponding co-factor.This process can continue recursively until the initial matrix is expressed in determinants of *2x2* matrices, from which the *nxn* determinant can be easily calculated through summing every component. In equation *(5)* C_{ij} is the co-factor of the matrix element a_{ij}.

$$|A| = \sum_{j=0}^{n} a_{ij}C_{ij} \tag{5}$$

The co-factor C_{ij}, is found using equation *(6)* where $|A_{ij}|$ is the determinant of matrix A after having removed the matrix elements from the i^{th} row and j^{th} column.

$$C_{ij} = (-1)^{i+j}|A_{ij}| \tag{6}$$

Finally the transpose can be defined in equation *(7)*. This is simply a reflection of a matrix along its leading diagonal, switching the columns to rows and vice-versa.

$$C_{ij}^T = C_{ji} \tag{7}$$

Equation *(8)* shows the standard matrix inversion formula used in this task where C^T is the matrix of cofactors for matrix A and $|A|$ is the determinant of matrix A.

$$A^{-1} = \frac{1}{|A|}C^T \tag{8}$$

Computation

A program was written to implement equation *(8)* to calculate the inverse of a square *nxn* matrix. The user is given control over the order of the matrix and whether to assign matrix elements manually or have them randomly assigned within the range of $-100 \leq x \leq 100$. For each user input checks are carried out to ensure a valid input type is used. Dynamic arrays were used to represent matrices due to the memory efficiency of the *malloc()* function. Functions for the determinant and matrix of cofactors were then called to the main function. The

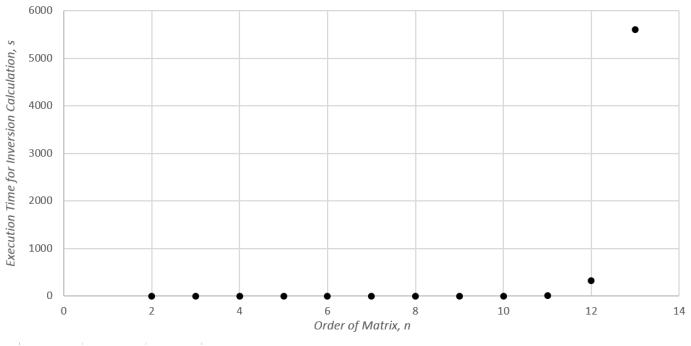

Figure 1: Plot of how the computation time for matrix inversion scales with the order of the matrix for the recursive determinant technique

recursive nature of the Laplace expansion required a function to be written that calls on itself within the function. This was used successfully in the determinant function, with checks put in place to ignore desired rows and columns when filling the minor matrices. The cofactor function was largely similar to the determinant one. It was extended through nested *for()* loops to iterate over every matrix element calling upon the determinant function to find the determinant of the reduced minor matrices. An extra check was needed to ensure the desired row and column were ignored correctly. Finally a simple transpose function was called to take the transpose of the matrix of cofactors. The main then contained the constituent components needed to compute equation *(8)* and the results are printed. In this program the matrices were passed to functions via the use of pointers which pointed to the array containing the matrix elements. The function types were set to void as their purpose was to just manipulate matrices inputted as arguments.

Results and Discussion

The execution time for the program was investigated first. The order of the matrix, n, was varied and the computation time for the inversion of a random matrix was recorded. Figure *(1)* shows the plot of how the execution scales with the matrix order. There is a dramatic increase in computation time at the inversion of a 13 x 13 matrix and the curve seems to have a rapid exponential growth.

However it was found that analysis with respect to execution time may be unfavourable. Different processors will give wildly different computation times due to processing power. Moreover external system processes will use up processing power causing variations on execution time even from the same machine. Another analytical technique is therefore needed. Finding the number of operations to compute the inverse is a far more reliable method of analysis. It can still be related to computation time as the number of mathematical operations for floating-points is proportional to computation time[2]. Finding the determinant of an *nxn* matrix through Laplace expansion involves performing $n!$ calculations, where each calculation is performed $(n-1)$ times leading to a operation count of $(n-1)n!$[3]. To find the total operation count for matrix inversion via the recursive determinant the operation count for the rest of the system is needed. This is found to be $n!-1$ leading to an overall operations count shown in equation *(10)*.

$$N = (n-1)n! + n! - 1 \tag{9}$$

Equation *(10)* accurately reveals the scaling of computational time with the matrix order. Further reading reveals the great inefficiency of the recursive technique as it often repeats calculations of minors making them redundant operations[3].

The programs robustness was increased with the introduction of error checks on user input values. This prevented the program from crashing after a user error with the use of an infinite *do()while()* that would break when the correct input was achieved. The accuracy of the program was also tested by comparing the programs results to those given by an online matrix inversion calculator[4] that uses the method of Gauss-Jordan elimination. The results from the program were printed out to 17 significant figures, the maximum precision of double floating-point numbers. It was found that the accuracy of the program to calculate the inverse of a 3x3 matrix was to 15 significant figures. The lack of full precision is likely from the accumulation of round-off errors over the course of the program. A 5x5 matrix was then tested for accuracy and was found again to have an accuracy of 15 significant figures. It was expected that it would be a lower accuracy as there would be a greater accumulation of round-off errors however this was not seen. For even larger order matrices the accuracy would certainly decrease due to this reason.

Problem 2: LU Decomposition and Singular Value Decomposition

LU Decomposition

LU (lower upper) Decomposition is the process by which an arbitrary matrix A can be expressed as the product of two matrices shown in equation *(10)*

$$A = L.U \tag{10}$$

$$A.x = (L.U).x = L.(U.x) = b \tag{11}$$

This decomposition allows the linear system, shown in equation *(11)*, to be represented as 2 triangular systems, shown in equations *(12)* and *(13)*.

$$L.y = b \tag{12}$$

$$U.x = y \tag{13}$$

These can be solved through simple forward and backward substitutions, making this method far easier and computationally preferential then the previous recursive method. Equation *(14)* shows the forward substitution, where α_{ij}'s are matrix elements from the lower triangular,b_i is the vector element from b and y_j is the vector element from y as shown in equation *(13)*.

$$y_i = \frac{1}{\alpha_{ii}}[b_i - \sum_{j=0}^{i-1} \alpha_{ij} y_j] i = 1, 2, ..., N-1 \tag{14}$$

Singular Value (SV) Decomposition

Another technique for the solution of linear systems is singular value decomposition. This operates on the principle that an arbitrary *mxn* matrix can be represented as the product of an *mxn* column-orthonormal matrix, an *nxn* diagonal matrix, with singular values, and the transpose of an orthonormal *nxn* matrix. In the tasks case we are dealing only with *m=n* and the form of the decomposition can be shown in equation *(15)*.

$$\begin{pmatrix} & A & \end{pmatrix} = \begin{pmatrix} & U & \end{pmatrix} . \begin{pmatrix} w_0 & & \\ & \ddots & \\ & & w_{n-1} \end{pmatrix} . \begin{pmatrix} & V^T & \end{pmatrix} \tag{15}$$

After decomposition the inverse of an *nxn* matrix can be trivial to find. The *nxn* matrices U and V are orthonormal so the inverse of these matrices is simply there transpose. The inverse of the diagonal matrix, W is simply the reciprocal of each element along the main diagonal, $\frac{1}{w_i}$ where *i=0,1,...,n-1*. The inverse of an arbitrary *nxn* matrix can then be expressed shown in equation*16*.

$$A^{-1} = V.[diag(\frac{1}{w_i}].U^T \tag{16}$$

Computation

To write a program to implement these algorithms it was necessary to use the GNU Scientific Library (GSL)[5]. This greatly simplified the programming with the only change being the type class for vectors and matrices and how they are handled. Once the user has inputted the matrix order and filled the matrix, functions from the GSL linear algebra library are called to first perform LU decomposition on the initial matrix, A, and then to find the inverse of A. The LU decomposition progresses via Crout's algorithm [6], this solves a specific set of equations by rearrangement of the equation order. This algorithm is tweaked to include the technique of pivoting, where a large matrix element is selected as a pivot and rows and columns interchange to fix the pivots position[6]. The function to find the inverse of the matrix A uses a special case of the GSL solving function for a general linear system. In this case an identity matrix is called and the GSL solve function is used to give the inverse matrix A^{-1}.

Programming the SV decomposition begins similarly with a user inputted gsl_matrix being defined. The SV decomposition shown in equation (*15*) is not the output of the decomposition function. Instead the output is the matrix U, the untransposed matrix V and the diagonal elements of the matrix W are stored in a gsl_vector. To get the form of equation(*16*) it was necessary to firstly call a GSL function to take the transpose of the matrix U. Then the reciprocal of each vector element is taken and transferred into a diagonal matrix W. This was done via an *if()* statement to only vector input elements on the main diagonal. Finally simple nested loops were used to perform a two part matrix multiplication to evaluate equation *(15)* giving the inverse matrix A^{-1}.

Results and Discussion

As before the matrix order was varied to find corresponding execution times for the inversion of a random square matrix. Figure *(2)* shows the results

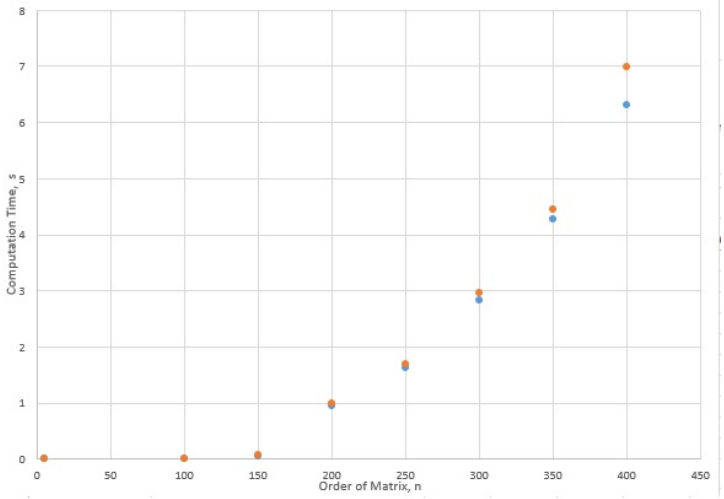

Figure 2: Plot of how the computation time for matrix inversion scales with the order of the matrix for the LU decomposition (blue) and the SV decomposition (orange)

from these investigations. LU decomposition and SV decomposition prove to be much more efficient at higher order matrices than the recursive technique. Again a discussion of the number of operations explains this best. For the LU decomposition the *nxn* matrix requires n^3 calculations to find the inverse matrix. These calculations arise from the n multiplications of n additions of rows for n iterations[7]. For SV decomposition the most costly step is the reduction of the matrix into a bidiagonal matrix, this carries an operations count of the order n^3. There is a second step, the SV decomposition of the bidiagonal matrix has operation numbers of the order of n^2. This is the cause of the slight increase in efficiency at higher orders seen with LU decomposition when compared to SV decomposition. However both techniques are far more preferential than the recursive technique.

The accuracy of each algorithm was tested when the input matrix became close to singular. The linear system solved is shown in equation *(16)* where k tends towards zero.

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & -1 \\ 2 & 3 & k \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 5 \\ 10 \\ 15 \end{pmatrix} \tag{17}$$

The value for k was iterated towards zero by a factor of 10 per iteration. The LU decomposition showed consistent results up to $k = 1x10^{-16}$ where machine precision is reached and k is rounded to zero. The results were $x = 0.0$, $y = 5.0$ and $z = 0.0$ and were compared with an online calculator. SV decomposition however gave very different results. By $k = 1x10^{-8}$ the results were beginning to vary as x was given as $x = 0.000001$. The accuracy worsened as k decreased. At $k = 1x10^{-14}$ the result was correct to zero significant figures, with the results $x = 7.129817y = -0.847594z = -3.001271$ The program crashed again at $k = 1x10^{-16}$ for the same reasons as before. Interestingly the SV decomposition is more revealing about how the the result at $k = 0$ is undefined by gradually losing accuracy as k approaches zero.

For future improvement it would be beneficial to introduce iterative improvement into the LU decomposition routine[7]. This method restores the machine precision of the solution of a linear system negating the accumulation of round-off errors. This would be crucial to improving the accuracy of the program especially when dealing with large linear systems. Iterative improvement works on the principle that the result found from the solution of a linear system is not represented by equation *(1)*, but instead by equation *(17)*, where δx is the unknown error and δb is the error in b caused by δx.

$$A.(x + \delta x) = b + \delta b \tag{18}$$

By subtracting equation *(1)* from equation *(17)* and solving for δb in equation *(17)*, appropriate substitutions can made to find equation *(18)*.

$$A.\delta x = A.(x + \delta x) - b \tag{19}$$

Equation *(18)* can be solved for δx as the entire right hand side is known. This method has order of operations of n^2 making it very applicable to any of the techniques used in this report.

Problem 3: Physics Problem

The final task was to apply one of these techniques to a physics problem. The problem involved finding the variation in the tension of a cable to suspend a camera for a football stadium. Before the program could be written the forces on the wire were resolved in the x,y and z directions. Using Pythagoras' theorem and trigonometric relations a set of simultaneous equations were constructed as seen in equations *(20-22)*. Where ϕ and θ are as shown in figure *(3)* and *(4)*. Figure *(3)* shows the top down view of the football pitch with the triangle representing the coverage of the camera. Figure *(4)* shows the z displacement of the camera and θ.

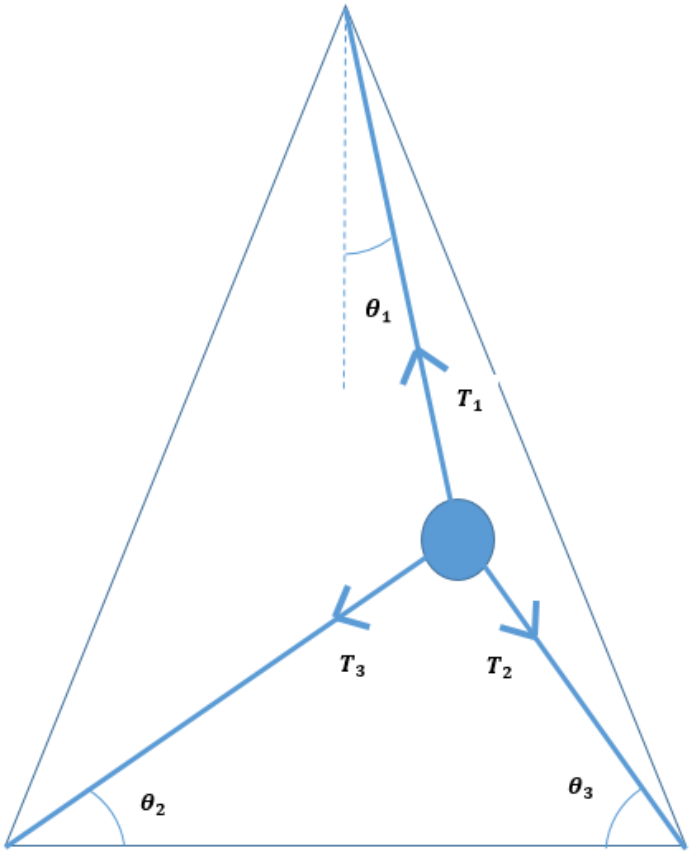

Figure 3: Birds eye view of the football pitch with the camera at an arbitrary position.

$$T_1 sin\theta_1 + T_2 sin\theta_2 + T_3 sin\theta_3 = 40g \tag{20}$$

$$-T_1 cos\theta_1 sin\phi_1 - T_2 cos\theta_2 cos\phi_2 + T_3 cos\theta_3 cos\phi_3 = 0 \tag{21}$$

$$T_1 cos\theta_1 cos\phi_1 - T_2 cos\theta_2 sin\phi_2 - T_1 cos\theta_3 sin\phi_3 = 0, \tag{22}$$

This can be expressed in the form of equation (*1*) as shown in equation *(23)*. The ¡math.h¿ library does not handle trigonometric functions well so it was then necessary to express equation (*23*) in terms of the camera's x and y coordinates. This gives the expressions in equations (*24-30*).

$$\begin{pmatrix} sin\theta_1 & sin\theta_2 & sin\theta_3 \\ -cos\theta_1 sin\phi_1 & -cos\theta_2 cos\phi_2 & cos\theta_3 cos\phi_3 \\ cos\theta_1 cos\phi_1 & -cos\theta_2 sin\phi_2 & -cos\theta_3 sin\phi_3 \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} 40g \\ 0 \\ 0 \end{pmatrix}, \tag{23}$$

The LU decomposition of this expression was then taken and the solver function was used to find values for T_1, T_2 and T_3. This process was iterated over the triangular area the camera could span and the results were written to a text file. A topographic plot of the football pitch could then be constructing showing areas of maximum tension.

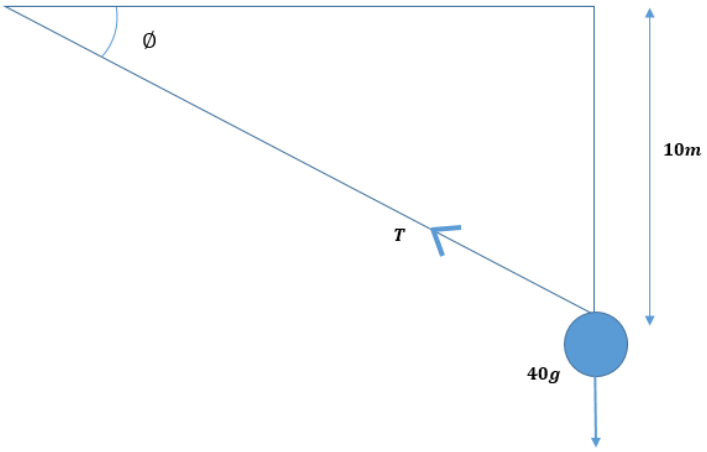

Figure 4: Cable shown in the vertical plane with the weight of the camera and θ labelled

$$\theta_i = sin^{-1}(10/r_i) \tag{24}$$

$$r_1 = \sqrt{x^2 + (80 - y)^2 + 100} \tag{25}$$

$$r_2 = \sqrt{(40\sqrt{3} + x)^2 + (40 + y)^2 + 100} \tag{26}$$

$$r_3 = \sqrt{(40\sqrt{3} - x)^2 + (40 + y)^2 + 100}, \tag{27}$$

$$\phi_1 = tan^{-1}(\frac{x}{80 - y}) \tag{28}$$

$$\phi_2 = tan^{-1}(\frac{40\sqrt{3} + x}{40 + y}) \tag{29}$$

$$\phi_3 = tan^{-1}(\frac{40\sqrt{3} - x}{40 + y}). \tag{30}$$

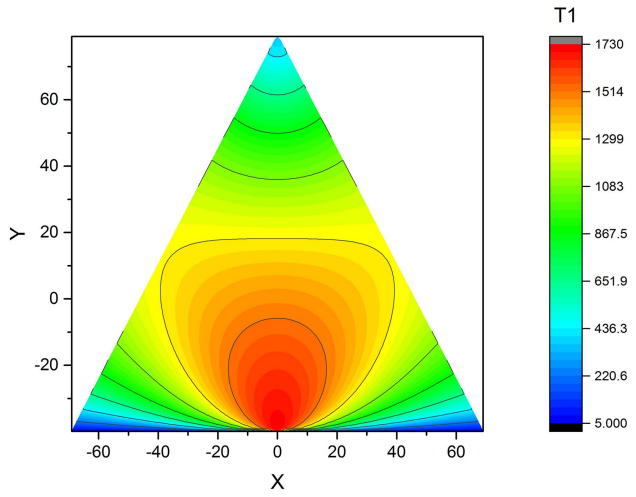

Figure 5: Topographic plot of the football pitch with respect to the tension T_1

Figure (*5*) shows the topographic plot of the football pitch for the tension T_1. However this graph is fundamentally flawed, the maximum was expected to be in two areas about mid way along the adjacent sides of the T_1 cable anchor point. This would be where the tension from the other two cables would be the strongest. Therefore there is a serious error in the way the program written progresses. Unfortunately due to time constraints this error could not be solved. However future work would be focused on resolving the force vectors again with greater care, prehaps dealing with a set of two simultaneous equations instead. They could be formed from the vertical and horizontal components of the tension alone, and implemented via SV decomposition. Even though there is more unknowns than equations SV decomposition can provide a non-unique solution in the form of a 'solution space'[7] which would be a good starting point to finding the true solution.

[1] Vein, Robert, and Paul Dale. "Determinants and their applications in mathematical physics." *Springer Science and Business Media* Vol.**134** (2006)

[2] Wyrzykowski, Roman, et al., eds. "Parallel Processing and Applied Mathematics: 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013" Revised Selected Papers. Vol.**8385** *Springer* (2014)

[3] Gentleman, W. M., and S. C. Johnson. "The evaluation of determinants by expansion by minors and the general problem of substitution." *mathematics of computation* **28.126** (1974): 543-548.

[4] Online Inverse Matrix Calculator, http://matrix.reshish.com/inverse.php, (Accessed 14/02/16)

[5] GNU Scientific Library (GSL) http://www.gnu.org/software/gsl, (Accessed 14/02/16)

[6] Forsythe, George E. "Algorithm 16: Crout with pivoting." *Communications of the ACM* **3.9** (1960): 507-508.

[7] Press, William H. "Numerical recipes 3rd edition: The art of scientific computing." *Cambridge university press* 2007.