Matrices and Linear Algebra

Sophia Wisniewska

INTRODUCTION

Matrices may be used to solve systems of linear equations [1]. An arbitrary set of simultaneous linear equations can be written as

$$M\underline{v} = \underline{b}$$

where M is the matrix of coefficients, $\underline{v}$ is the vector of unknown variables and $\underline{b}$ is the known vector of constants. By finding the inverse of M, the unknown variables can be determined. This is known as the analytical inversion method (AIM), and in this set of programs a coding routine shall be written in order to investigate the behaviour of this method for solving systems of linear equations. In addition, built-in python functions shall be used to write alternative methods to solve these equations, notably LU decomposition (LUD) and Singular Value Decomposition (SVD), and these methods compared against AIM. The solving of a system of linear equations is then put into the context of a real-life physics problem.

METHODS

AIM

A routine was written to invert an arbitrary square NxN matrix using the standard inversion formula

$$M^{-1} = (1/detM)C^T$$

where M^{-1} is the inverse matrix, detM is the determinant of the original matrix M and C^T is the transpose of the matrix of cofactors.

LUD and SVD

LU and SVD are both alternative methods of solving a system of linear equations. LUD involves decomposing original matrix M into lower (L) and upper (U) triangular matrices [2], while SVD is the factorization of M into the product of 3 matrices $= UDV^T$, where the columns of U and V form orthonormal vectors, and diagonal matrix D contains the singular values [3]. SVD is more versatile than AIM and LUD as it can be used to solve MxN systems as well as NxN [3]. It also tends to be more reliable in solving close-to singular matrices.

The built-in functions *linalg.lu_factor* and *linalg.lu_solve((LU, P),b)* were used to find $\underline{v}$ using the LUD method, and the functions *linalg.svd(M)*, *linalg.diagsvd(s,N,N)* and *linalg.inv(s)* were used to find $\underline{v}$ using the SVD method. These functions were taken from the *scipy* library.

RESULTS AND DISCUSSION

AIM is only executable up to N=9 as beyond this value, the console displays the error "RuntimeWarning: overflow encountered in long_scalars". It was found that the error is due to variables having a finite storage space. Originally this error occurred for matrices beyond N=5 but this was resolved by changing the datatype attribute of the matrix from "int32" to "int64". Since "int64" is the datatype of maximum storage space in Python, the inverse of matrices beyond N=9 cannot be computed by this program.

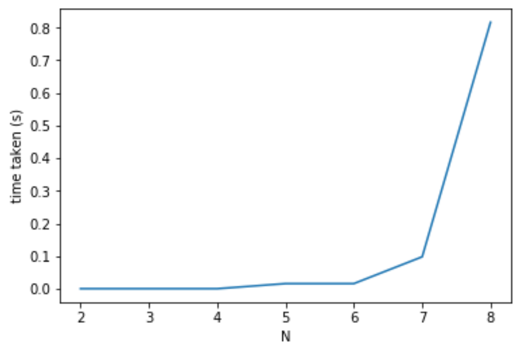

FIG. 1. Graph of the time taken to solve the linear system of equations using AIM against the number of rows in the matrix.

The runtime of AIM appears to increase exponentially with the number of rows as shown in fig.1, which is expected as having more rows increases the number of iterations in the functions detnxn and cofactorsnxn. This cannot be avoided, but measures can be taken to improve the efficiency of the program, for example removing redundant code- the generateempty(n) function simply provides an N dimensional array to which cofactorsnxn(M,n) appends its elements- the code could have been written such that generateempty(n) isnt needed, and the cofactor matrix is simply constructed during the appending of the elements. In addition, hardware changes such as investing in a computer with a better processor would improve the efficiency.

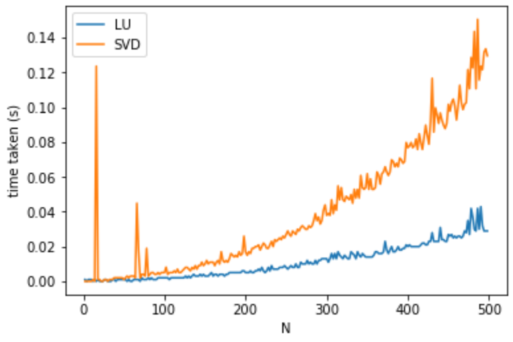

FIG. 2. Graph of the time taken to solve the linear system of equations using LUD and SVD against the number of rows in the matrix.

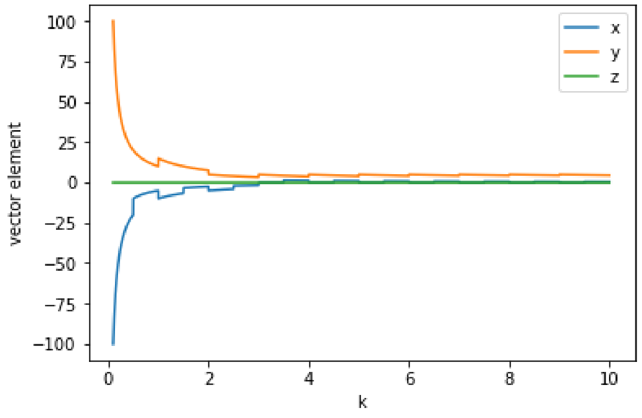

FIG. 3. Graph of the solutions to the system of linear equations against k using AIM. Despite the z solution being consistently accurate, x and y tend to $+\infty$ and $-\infty$ respectively as k tends to 0.

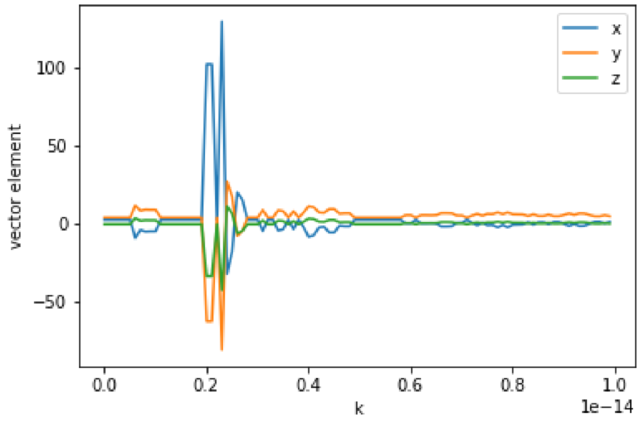

FIG. 4. Graph of the solutions to the system of linear equations against k using SVD.

In fig.2, the runtimes of LUD and SVD also increase exponentially, but over larger N compared to AIM. In fact, LUD appears to be the most efficient method as its increase in runtime over a region of N from 2 to 500 is smaller than that of SVD as shown in fig.2. The runtimes can also be seen to fluctuate in both figs. 1 and 2, with fluctuation being more prevalent in fig. 2 as more iterations were performed. Fluctuation in runtime of all programs is to be expected as the computer processor may be running multiple processes at once, for example, from background applications, so has to share the computing power, meaning each run of the programs will be different. There appears to be a couple of abnormally large fluctuations in runtime occuring around low N for SVD, however these have been ruled out as random after executing the program several times to see whether these fluctuations prevail.

The behaviour of AIM when the matrix is close to singular (close to having 0 determinant) was investigated using the following system of linear equations:

$$x + y + z = 5$$
$$x + 2y - z = 10$$
$$2x + 3y + kz = 15$$

for small k, where k is also equal to the determinant of the matrix.
It was found that LUD produced the correct answer of x=0, y=5, z=0 consistently for all values of k. From fig.3, the solutions from AIM converge to the correct values for large k, but the method breaks down for small k (notably $k < 1$). This is due to the matrix tending towards singular, as k is also equal to the determinant of the matrix. As k tends to 0, the determinant of M tends to infinity, thus AIM is no longer a suitable method to solve the system of equations. It can also be seen from fig.3 that the apparent exponential function contains periodicity, perhaps best described as in the form of an envelope square wave. The reason for this converging periodic function is unknown, but the regularity of it suggests it must be more than random fluctuation. In addition, the value of z does not change with k unlike x and y do, and is probably due to the z elements not being affected by k when the new elements are calculated from the functions involved in AIM- when co-factors are calculated, the ith row and jth column are deleted- in this case, the column containing all z elements, including k, is deleted, so all the z elements are independent of k.

Fig. 4 displays a spike in the solutions around 0.2×10^{-14} using the SVD method. It is unusual that the error drops back down beyond this spike as k tends to 0. Normal fluctuation can be observed elsewhere, but the cause for this abnormally large fluctuation in all the vector components is unknown.

Fig.5 shows one of the two surface plots for the tension in the front two wires as a function of position. Maximum tension was found to be 2589.7 N (1d.p.), occurring at x=7.5m and z=7.9m for both wires. This agrees with back-of-the-envelope calculations, and also makes intuitive sense as the

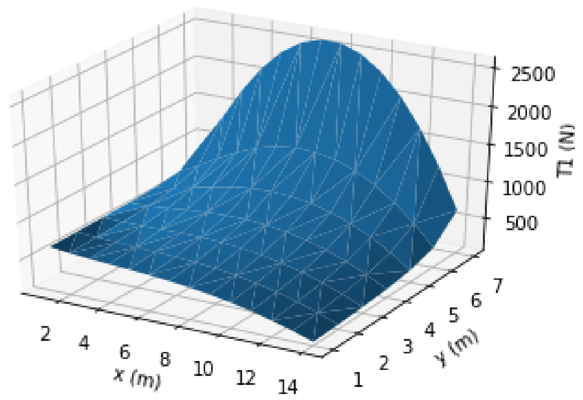

FIG. 5. Surface plot of the tension, T1 in the first wire as a function of position when considering the 2 front wires only, confined to the plane y=0.

acrobat is at maximum height, so this is where maximum tension is as the wires are horizontal and level with the drums- they are pulled 'taut'. The reason for z=7.9m is due to the precision of the increments- increments of 0.1 were used in the for loop, hence the deviation from the actual drum height of 8m. However, perhaps this value of 7.9m is more realistic as it would be very difficult to achieve a large enough wire tension that would result in the acrobat being at the exact height of the drums.

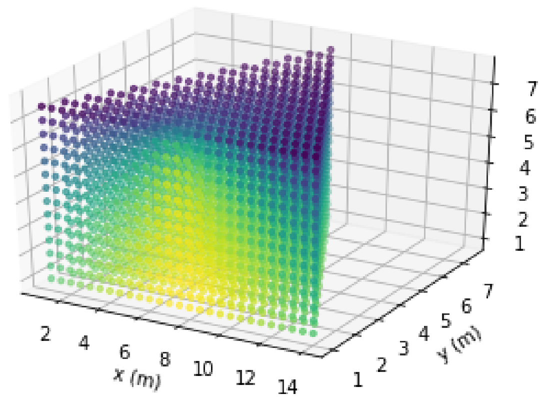

FIG. 6. Scatter plot of the tension, T3, in the 3rd wire for the system involving 3 wires. Low to high tension is from light to dark.

Fig. 6 shows that the tension in the 3rd wire is minimum in the vertical plane at y=0, since in this region the acrobat is being supported independently by the two front wires, and the 3rd wire is 'slack'. T3 increases with y, as the acrobat's weight is increasingly supported by the third wire as they are pulled towards the vertical plane at y=8 which contains the 3rd wire's drum. Fig.7 shows that the tension in the 2nd wire is at a maximum when the acrobat is furthest away from the 2nd drum in the x and y plane, and in fact the tension graph

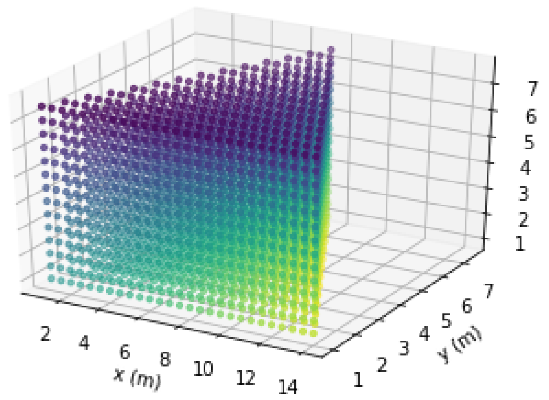

FIG. 7. Scatter plot of the tension, T2, in the 2nd wire for the system involving 3 wires. Low to high tension is from light to dark.

for T1 is an exact mirror of fig.7 along x, as the two front wire drums are on opposite sides to each other in the system. The fact that the dark region resides at approximately z=8 for both figs. 6 and 7 clearly shows that maximum tension occurs at the greatest height achievable (i.e.: level with the drum height).

The position of max tension was found to be x= 4.0 m y= 4.0 m z= 7.0 m; x= 11.0 m y= 4.0 m z= 7.0 m and x= 7.5 m y= 1.0 m z= 7.0 m for the 1st, 2nd and 3rd wire respectively. The magnitudes of the maximum tension were 534.1N for the 2 front wires and 678.1N (1d.p.) for the 3rd wire. These position values are intuitive, as with the 3rd wire involved it is agreeable that the 2 front wires will be at their maximum tension a quarter a way along the x axis from their drum positions, and that in the y and z plane they will be in the middle. These position values are however rounded to the next 0.5, as the iteration precision was set as 0.5 to reduce runtime. Furthermore, the range in the for loops iterating over x,y, and z were made to iterate up to 14, 7 and 7 respectively rather than 15, 8, and 8 in order to avoid division by 0.

CONCLUSIONS

The analysis of program runtime against number of rows and error in vector elements as the matrix tends to singular shows that LUD is the best method for solving systems of linear equations in computer programming. However, it is only able to solve systems which involving square matrices- in this case SVD must be used. SVD is too a suitable method, however it has an unusual spike in error at a certain value of k that does not follow any particular trend, and its runtime is significantly longer than that of LU for large-scale N. The spike in error goes against SVD being best for solving close-to singular matrices. AIM proves to be the worst method as it involves several iterations over the same functions, and Python's storage capabilities are limited. The function written to find maximum tension in 2D proves to be accurate with the position at a precision of 0.1m. The scatter plots of the 3d system of wires demonstrates well the intensity distribution of force, however precision could have been increased by iterating over smaller increments.

REFERENCES

[1] *Computational Physics 301, Lecture 1*, J. Brooke, slide 26

[2] *LU-Decomposition Computerized Method to Solve Linear Programming Problems*, Abdulraheem M.Z. and Mohammad K, Department of Mathematics, University of Business and Technology, Jeddah, Saudi Arabia (2018), Vol 7(2)

[3] *Essential Numerical Computer Methods*, M.L. Johnson, Elsevier Inc. All rights reserved (2010), p.91