Project 1 - FYS3150

Markus Borud Pettersen

August 2019

Abstract

By investigating numerical solutions to the Poisson equation in one dimension, we find that the second derivative, and indeed the solution to the Poisson equation, can be found using a simple matrix equation, where the components of the approximate second derivative constitute a square, tridiagonal matrix. In order to solve the matrix equation, a general Gaussian elimination method for tridiagonal matrices was developed. In addition, an algorithm for the specific case of the second derivative matrix was also designed, and the execution times of these algorithms were compared. Finally, the Gaussian elimination methods were compared with a pre-existing lower-upper decomposition algorithm, both in terms of results, and execution speed. For the general and specific algorithms, execution speeds, and relative errors, were found to be comparable for different numbers of calculation steps. The lower upper decomposition produced equivalent results to the general algorithm, but could not be used for larger numbers of calculation steps, due to memory shortages.

Introduction

Many interesting phenomena can be modelled using the second derivative of some physical quantity, be it the acceleration of an object due to gravity, or the probabilistic behaviour of certain quantum systems. If we restrict ourselves to spatial derivatives, however, we can still study rich physical systems, many of which obey the Poisson equation, a partial differential equation. In one dimension, it takes on the rather appealing form

$$-\frac{d^2u(x)}{dx^2} = f(x), \quad x \in (0,1), \tag{1}$$

where u is some physical quantity, x a spatial coordinate, and $f(x)$ some known function describing the second derivative. In this discussion, we will only consider Dirichlet boundary conditions $u(0) = u(1) = 0$. In one dimension, the Poisson equation may for instance describe the electrostatic potential of a conductor, for a charge distribution $f(x)$. As one might imagine, however, analytical solutions of (1) are few and far between, especially when systems become more complex. To remedy this, we will attempt to create numerical approaches to solving it. To do so, we first need to discretize our equation, and our variable x. In other words, we evaluate x at points $x_i = ih$, where i is an index, and h the stepsize. We can approximate (1) as

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = g_i, \quad i = 1, 2, ..., n,$$

where $v_i = u(x_i)$ and n is the number of points for which we perform calculations. We will use a constant stepsize, given by $h = \frac{1}{n+1}$.

We can easily rewrite the above equation and get

$$-v_{i+1} + 2v_i - v_{i-1} = f_i, \tag{2}$$

where we define $f_i = h^2 g_i$, for convenience. There is one important takeaway from this way of writing our equation; it is not only a linear equation, it is a set of linear equations, as each index gives a different equation. We know that we can solve sets of linear equations by arranging them

in a matrix, and then applying gaussian elimination. By inspection, we can find that a complete, discretized approximation to (1) can be written as

$$\begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 \\ 0 & \dots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \dots & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}. \tag{3}$$

Note that the values of v_0 and v_{n+1} are given by the boundary conditions as zero.

The matrix in (3) is in fact a tri-diagonal matrix, as all elements are distributed on and around the diagonal. As one can imagine, Gaussian elimination is more straight forward for a matrix like this, as most elements are zero. If we assume only unique (non-zero) elements along the diagonal, solving (3) can actually be decomposed into two steps: One forward pass, where we eliminate the elements below the diagonal, and one reverse substitution, where we actually solve for the variables we are looking for. In this general case, we will denote elements below the diagonal by a_i, diagonal elements by b_i, and elements above the diagonal by c_i, where i is an index. In order to remove an element a_i in row number i, we can simply subtract the above row, multiplied by a factor a_i/b'_{i-1}. Doing so, we end up with updated elements

$$\begin{aligned} b'_i &= b_i - a_i c_{i-1}/b'_{i-1} \\ f'_i &= f_i - a_i f'_{i-1}/b'_{i-1}, \end{aligned}$$

where the $'$ indicates an updated value. Note that for $i = 1$, $b'_1 = b_1$ and $f'_1 = f_1$. When the forward pass is complete, we can begin to back-substitute. We do so by noting that we, after the elimination of all a_i, are left with

$$b'_i v_i + c_i v_{i+1} = f'_i,$$

which is easily rearranged, such that

$$v_i = (f'_i - c_i v_{i+1})/b'_i,$$

which gives us our solution. Using the boundary condition $v_{n+1} = 0$, we also find that $v_n = f'_n/b'_n$, which should allow us to solve the entire system. By counting, we find that we should be able to use a grand total of $8n$ floating point operations (given that we calculate a_i/b'_{i-1} only once), to solve it.

Now that we have a general algorithm, we may begin to optimize it for the very specific case shown in (3), where all elements along diagonal elements are twos, and the rest either negative ones, or zeros. In other words, $b_i = b_2 = ... = b_n = 2$, and $a_1 = c_1 = a_2 = c_2 = ... = a_n = c_n = -1$. Substituting these values in our expression for the updated elements, we get

$$\begin{aligned} b'_i &= 2 - 1/b'_{i-1} \\ f'_i &= f_i + f'_{i-1}/b'_{i-1}, \end{aligned}$$

but we can simplify things further, by noting a pattern: Starting at $i = 1$, we have that $b'_1 = b_1 = 2$, and subsequently that $b'_2 = 2 - 1/2 = 3/2$. Going a step further shows that $b'_3 = 2 - 2/3 = 4/3$, and a pattern starts to emerge, namely that

$$b'_i = \frac{i+1}{i},$$

in the special case. We will not attempt to prove this result, but it could possibly be proved quite easily by means of induction, for instance. Finally, we get that

$$v_i = (f'_i + v_{i+1})/b'_i,$$

which, by counting, should take a total of $6n$ floating point operations. We can, however argue that we could pre-calculate the values for b'_i and form a lookup table for many values of n, which reduces the number of floating point operations to 4.

Methods

Both the general and simple algorithms for Gaussian elimination were implemented in c++. To reduce memory load, only the tridiagonal elements were stored and used for computations. To allow for different matrix sizes, and number of calculation steps, dynamic memory handling was used. A source term, $f(x) = 100e^{-10x}$, was taken to be a test case for the algorithms. For this choice of $f(x)$, the Poisson equation with Dirichlet boundary conditions has the analytical solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$ [1]. Both algorithms were then tested using this source term, for $n = 10$, 10^2, 10^3, 10^4 and 10^5.

For a given number of calculation steps n, the maximum relative error between the approximation v_i and the analytical solution u_i was logged as

$$\epsilon_{max} = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right). \tag{4}$$

As an additional verification, an external lower upper (LU) decomposition algorithm [2] was used to solve the matrix equation in (3), and the difference between solutions compared. For the general, simple and LU decomposition algorithms, execution times were also logged for each value of n.

Results

Table 1 shows the execution time for the simple and general Gaussian elimination methods, as well as the LU decomposition method, for different values of n. Strikingly, the execution time for the LU decomposition is several orders of magnitude greater, than for the general and simple algorithms. For values of n greater than 10^3, the test computer began to run out of memory when attempting the LU decomposition, and execution time became indefinite. For the general algorithm, we can see that the maximum ($\log_{10}$) relative error is greatest for lower values of n, and decreases with increasing n. In fact, the logarithmic relative error is approximately linear, decreasing by around 2, for every order of magnitude n is increased. Also, the relative error was greatest early on in the simulation, for all test cases. Also computed was the average absolute difference between the solutions found using the LU decomposition, and the general and simple algorithms. For the general algorithm, this difference was found to be zero, while for the simple algorithm, it was on the order of 10^-16.

The execution times between the simple and general elimination algorithms were for the most part identical. Note that the execution time was only listed with one significant figure, as the results varied greatly between each execution.

Fig. 1 shows the calculated approximation of the function $h^2 f(x)$ using the general elimination algorithm for different values of n, alongside the analytical solution evaluated at $n = 1000$ points. As is obvious with the $n = 10$ case, the result is only shown at the points where calculations are actually performed, as the boundary points are trivially equal and zero. We can note that the $n = 100$ and $n = 1000$ cases are hard to tell apart from the analytical solution.

Discussion

Both from Fig. 1 and Table 1, we can see that the error made using the general algorithm can be made quite small, given a sufficiently large n. Also, the small error suggests that the algorithm is in fact performing as it should, which is further strengthened by the fact that the average difference between the LU decomposition and general elimination solutions, is zero. While not zero, the average difference between the LU decomposition and the simple algorithm is also vanishingly small, at approximately 10^{-16}. It is not entirely clear what causes this difference, but it seems to suggest that the LU decomposition and the general algorithm is at some level numerically equivalent. The simple algorithm, on the other hand, works slightly differently from the general algorithm, in that it possibly avoids subtraction of terms, which might help to avoid numerical round-off error, especially if the numbers involved are very small. However, in the implementation of both algorithms, double precision values were used, which should help to

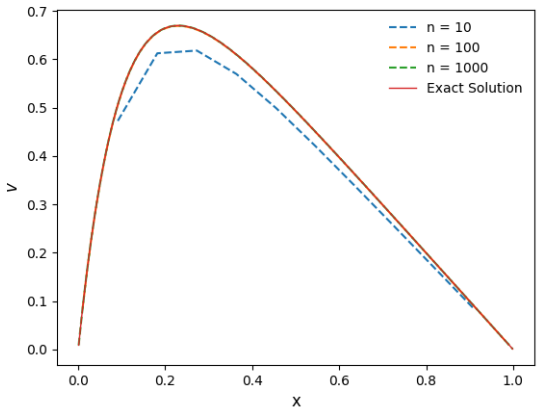

Figure 1: Approximate solutions to the Poisson equation for the source term $f(x) = 100e^{-10x}$, for different numbers of calculation steps n, shown alongside the analytical solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$ in the interval $x \in (0, 1)$.

Table 1: Execution times for the general, simple and LU decomposition algorithms, when applied to the source term $f(x) = 100e^{-10x}$ with Dirichlet boundary conditions, for different numbers of calculation steps, n. Note that for large values of n, the LU decomposition could not be determined due to memory errors. *Also recorded is the computed maximum relative error ($\log_{10}$), but only for the general algorithm.

Execution time [μs]			Matrix size	Max Relative Error*
General	Simple	LU	n	ϵ_{max}
2	1	7	10	-1.18
3	3	10^3	10^2	-3.08
20	20	10^6	10^3	-5.08
200	200	-	10^4	-7.08
2000	2000	-	10^5	-8.84

safeguard against round-off errors of this kind. Also, the relative errors of the general and simple algorithms were quite similar, but the absolute maximum relative error made using the simple algorithm was slightly smaller in all cases tested, on the order of 10^{-8}-10^{-12} ($\log_{10}$). Even though this difference is minuscule, it might be an interesting case for further study to see if this difference is actually due to round-off errors or something related to it.

It is also interesting to see that the execution time for both the general and simple algorithms increases approximately linearly with the number of calculations, as we would expect. It is, however unexpected that the general and simple algorithms take more or less the same amount of time to execute, when we would expect the simple algorithm to be around 50 % faster, at least in terms of the number of floating point operations. It is not clear what causes the execution time difference, but it might hint at a bug in the implementation, or some unknown bottleneck in the program execution. It might be that the pre-calculation of the diagonal elements, which is done at runtime, for simplicity, forces a conversion from integer to float division, which could add another floating point operation to the simple algorithm. This would bring the two pretty close, and possibly make any difference in execution time hard to pin down, especially when the execution time fluctuates between runs. One possible way to combat this, would be to run the

program many times, and average the execution times over many runs.

We can, however note that the execution time of the LU decomposition algorithm scales as approximately n^3, which is around what we expect [3]. The reason the LU decomposition algorithm fails for large n seems to be due to memory shortages, as the full matrix representation requires the storage of n^2 numbers, while the simple and general elimination algorithms more or less only store the elements on and around the diagonal, meaning we only have to store something on the order of n numbers.

Conclusion

By investigating the one-dimensional Poisson equation, we have found that its solution can be approximated numerically, and that the numerical problem may be represented by a matrix equation. By taking advantage of the fact that this matrix is tridiagonal, a Gaussian elimination algorithm was implemented, and tested for a source term $f(x) = 100e^{-10x}$, which, given Dirichlet boundary conditions, has the solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. In addition to an algorithm for general tridiagonal matrix elimination, a specialized algorithm was also created, effectively implementing the second derivative in matrix form.

The execution times of these algorithms were found to be quite similar, even though the specialized algorithm should have performed somewhat faster. While the reason for this is not fully understood, it would make an interesting case for further study. For the general algorithm, the maximum relative error ($\log_{10}$) was computed for different numbers of calculation steps n. This relative error was found to decrease somewhat linearly with number of steps n, and was similar both for the general, and the specialized algorithms.

These algorithms were also compared with an external LU decomposition algorithm, for verification purposes. The LU decomposition and general algorithms seemed to produce equivalent result, but were slightly different from those of the specialized algorithm, the average absolute difference being on the order of 10^{-16}. However, the LU decomposition algorithm proved unusable for values of n greater than 10^3, due to over-use of memory associated with storing the entire tridiagonal matrix.

Collectively, these findings provide a valuable basis for the use of various numerical methods for solving the Poisson equation, and related problems. It also helps to shed light on the importance of sensible memory handling, and selection of solution strategies for such problems, while also showing the connection between floating point operations, and program execution time.

References

[1] M. Hjort-Jensen, "Project 1", Computational Physics I FYS3150 - Project 1, Oslo: Department of Physics, University of Oslo, Aug. 22, 2019

[2] M. Hjort-Jensen, "ComputationalPhysics cpplibrary", GitHub, 2016. [Online]. Available: https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Programs/LecturePrograms/programs/cppLibrary/lib.cpp. [Accessed: 08- Sep- 2019].

[3] M. Hjort-Jensen, "Gaussian Elimination", in Computational Physics - Lecture Notes Fall 2015, Oslo: Department of Physics, University of Oslo, 2015, p. 173.

Additional Material

All source code is freely available from https://github.com/markusbp/fys3150/tree/master/project1

// Runtime example for code /* Execution time for simple algorithm: 5.401e-06 s for n=100 Maximum relative error (log10): -3.08803683155941 Execution time, LU decomposition: 0.001802405 Average absolute difference between LU and Gaussian elimination methods: 4.58690377347359e-15 */