Two methods of solving linear second-order differential equations with Dirichlet boundary conditions

Anna Lina P. Sjur and Jan-Adrian H. Kallmyr

September 9, 2018

Abstract

State problem. Briefly describe method and data. Summarize main results.

1 Introduction

Physics is a field concerned with the behaviour of nature, and nature is everchanging. It is therefore no surprise that differential equations appear everywhere in physics. From global climate dynamics to statistical mechanics, what we find is that differential equations, often many and coupled, are required to explain or model the phenomena. For such large models, efficiency is important, as we would, for example, like to have timely weather forecasts. One way to make a model more efficient is by using an efficient algorithm for solving differential equations.

In this article, we compare two different numerical methods of solving linear second-order differential equations with the Dirichlet boundary conditions. To do this, we will solve the one-dimensional Poisson's equation:

$$\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} = -4\pi r\rho(r). \tag{1}$$

In the methods section, we develop an approximation for the 2nd derivative to the 2nd order. We will then solve eq. 1 numerically, using gaussian elimination and lower-upper decomposition. As for the former, we will further specialise it to solve eq. 1 more efficiently. Next, in the results section we present the comparison between our numerical solutions and the analytical solution, as well as the error. We then compare the efficiency of all three algorithms, and finally, in the discussion section we will consider the advantages and disadvantages of each algorithm, and discuss their uses.

2 Methods

We would like to solve eq. 1 numerically. Generalising the equation, we get

$$-\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = f(x), \tag{2}$$

where we have assumed that $\rho \propto \frac{1}{r}e^{-r}$ and let $r \to x$, $\phi \to u$. Summing the backward and forward Taylor expansions of $u(x)$ and discretising the equation for n integration points, we get:

$$\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} + \mathcal{O}\big(h^2\big) = f_i, \tag{3}$$

where $h = \frac{1}{n+1}$. Using the Dirichlet boundary conditions $v_0 = v_{n+1} = 0$ and only considering $x \in (0, 1)$, we can rewrite the equation as a set of linear equations, represented as the following matrix equation:

$$A\mathbf{v} = \mathbf{d}, \tag{4}$$

where

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \ddots & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix},$$

is a tridiagonal matrix and $d_i = h^2 f_i$. Having the equation in this form, we can use linear algebra to solve the set of linear equations, and obtain the 2nd derivative of $u(x)$.

2.1 Gaussian elimination

The fact that A is tridiagonal, means that we can develop a general algorithm to solve the equations by the method of Gaussian elimination. If we generalise our matrix

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_1 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_2 & b_3 & c_3 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \ddots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & 0 & a_{n-1} & b_n \end{bmatrix},$$

and row reduce A to become upper-triangular (see appendix A.) This yields us the following relations:

$$\widetilde{b_i} = b_i - \frac{a_{i-1}}{\widetilde{b}_{i-1}} c_{i-1}, \quad \widetilde{d_i} = d_i - \frac{a_{i-1}}{\widetilde{b}_{i-1}} \widetilde{d}_{i-1}, \quad (5)$$

where $\widetilde{b_1} \equiv b_1$ and $\widetilde{d_1} \equiv d_1$. Performing the matrix-vector multiplication in eq. 4, we get the following relations from the second-to-last and last row:

$$v_i = \frac{d_i - c_i v_{i+1}}{\widetilde{b_i}}, \quad v_n = \frac{\widetilde{d}_n}{\widetilde{b_n}}. \quad (6)$$

With these expressions, we can construct our general algorithm as follows:

initialise **v**, **a**, **b**, **c**, **d**;
for $i = 2, \dots, n$ **do**
  $b_i = b_i - \frac{a_{i-1}}{b_{i-1}} c_{i-1}$;
  $d_i = d_i - \frac{a_{i-1}}{b_{i-1}} c_{i-1}$;
$v_n = \frac{d_n}{b_n}$;
for $i = n-1, \dots, 1$ **do**
  $v_i = \frac{d_i - c_i v_{i+1}}{b_i}$;

This algorithm has a total of 8n FLOPS.

If we now take into account that our matrix has only twos along the diagonal, and -1 above and below the diagonal, we can make a specialised algorithm. Plugging in theses values in the expression for $\widetilde{b_i}$ in equation 5 gives us the following:

$$\widetilde{b_i} = 2 - \frac{1}{\widetilde{b}_{i-1}}, \quad \widetilde{b}_1 = 2 \quad (7)$$

We can prove by induction that this is the same as:

$$\widetilde{b_i} = \frac{i+1}{i}, \quad i \geq 1 \quad (8)$$

Since $\widetilde{b_i}$ is only a function of i we can now set up $\widetilde{\mathbf{b}}$ in the initialisation. Plugging in the values in the remaining expressions gives:

$$\widetilde{d_i} = d_i + \frac{\widetilde{d}_{i-1}}{\widetilde{b}_{i-1}}, \quad v_i = \frac{\widetilde{d_i} + u_{i+1}}{\widetilde{b_i}} \quad (9)$$

where $v_n = \frac{\widetilde{d_n}}{\widetilde{b_n}}$. This gives us the following specialised algorithm, with a total of 4n FLOPS (not counting the initialisation of $\widetilde{\mathbf{b}}$):

initialise **v**, **b**, **d**;
for $i = 2, \dots, n$ **do**
  $d_i = d_i + \frac{d_{i-1}}{b_{i-1}}$;
$v_n = \frac{d_n}{b_n}$;
for $i = n-1, \dots, 1$ **do**
  $v_i = \frac{d_i + v_{i+1}}{b_i}$;

2.2 LU-decomposition

Let A be a square matrix, then we can decompose it in the following form

$$A = LU, \tag{10}$$

where L is a lower triangular matrix and U is a upper triangular matrix. For further conditions on A, see [1]. This lets us rewrite eq. 4 in the form:

$$(LU)v = d, \tag{11}$$

where we let

$$y = Uv, \tag{12}$$

and rewrite eq. 11 to

$$Ly = d. \tag{13}$$

To solve for v then we need to first solve eqs. 13 for y and then substitute y into eq. 12 where we can then solve for v, yielding us the solution to eq. 4.

This then is our general LU-decomposition method, and it has around $\frac{2}{3}n^3$ flops. We could elaborate further on the finite difference details hidden away in the notation, but this algorithm is implemented in armadillo (see the Programming technicalities section), and so most of these details aren't handled by us.

2.3 Error

We define the error for each mesh point as:

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{14}$$

where $u_i = u(x_i)$ is the analytical solution given by:

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{15}$$

2.4 Comparing CPU time

When comparing the CPU run time for the different algorithms, we time only the forward and backward substitution. We do not take into account the initialisation of the different variables. Each run gives a different run time, so the average of ten runs are presented in this report.

2.5 Programming technicalities

We implement our algorithms and methods in C++. For our Gaussian elimination algorithm we use only the standard library to declare and operate arrays. For the LU decomposition algorithm we use the armadillo library with LAPACK to declare matrices and perform the LU-decomposition as well as solving equations 13 and 12.

3 Results

Let's start with the solution to equation 4. Our general matrix solver gave the results shown in Figure 1 for n=10, n=100 and n=1000. The analytic solution is also shown. For n=1000 the numerical and analytical solutions are so close to each other that we can not distinguish them in the plot, even after zooming in with a factor of 100. Both the specialised algorithm and the LU decomposition algorithm gave the same result as the specialised algorithm, see Figure 3 in the appendix.

We wanted to get an idea about the efficiency of the different algorithms and how they compare. Table 1 shows the ratio between the CPU time for the general algorithm and the specialised algorithm, as well as the ratio between the LU decomposition algorithm and the specialised algorithm for different values of n. We see that the specialised algorithm is between 40% to 100% faster than the general algorithm, where the difference appears to de-

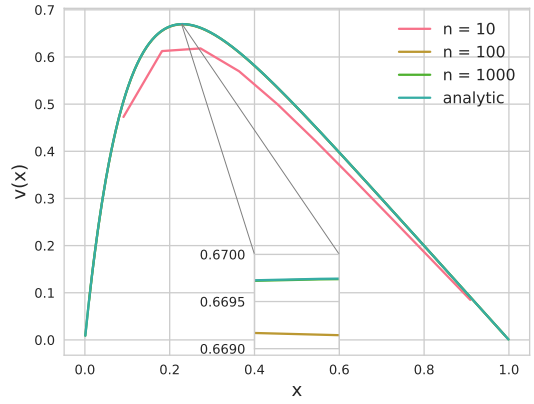

Figure 1: The numeric solution using different numbers of steps and the analytic solution. The window shows a section of the plot zoomed in with a factor of 100.

crease as n increases. On the other hand, the ratio between the LU decomposition algorithm and the specialised algorithm increases rapidly with n. For $n = 10^4$ the specialised algorithm is more than a million times faster than the LU decomposition. As we were not able to run the program for $n > 10^4$, we have no values to present in this domain.

$\mathbf{n}$	$\mathbf{t_g/t_s}$	$\mathbf{t_{LU}/t_s}$
10	2.08	3.70
10^2	1.89	$1.00 \cdot 10^2$
10^3	1.48	$1.05 \cdot 10^4$
10^4	1.43	$1.18 \cdot 10^6$
10^5	1.39	-
10^6	1.41	-
10^7	1.39	-

Table 1: Ratio between CPU time for the general algorithm ($\mathbf{t_g}$), the special algorithm ($\mathbf{t_g}$) and the LU decomposition algorithm ($\mathbf{t_{LU}}$) for different matrix sizes ($\mathbf{n}$). The LU decomposition crashed for $\mathbf{n}$ greater than 10^4.

Lastly we wanted to assert the error in our numerical solutions. Figure 2 shows the maximum relative error in our specialised algorithm as a function of n with a logarithmic scale. We see that the error decreases with n until $n = 10^6$, and then increases.

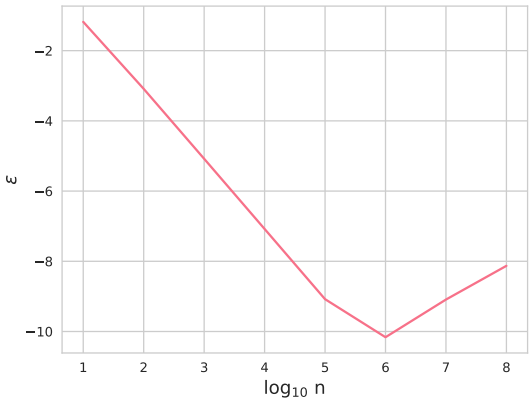

Figure 2: The maximum error in our specialised matrix solver as a function of the number of steps/matrix size on a logarithmic scale.

4 Discussion

Having presented the different methods and their relative efficiency and accuracy, we would like to discuss their uses as well as their boundaries. We see that for our problem of solving eq. 2, we could construct a very specialised and efficient algorithm because we used a 2nd order central approximation to the 2nd derivative. This let us reduce the number of flops by a half, and our computer should, with an optimised CPU, only have to do half the work. In terms of accuracy, we found no significant difference between the algorithms, and so this also means that we get the most accuracy per CPU time with our specialised algorithm up until $n=10^{-6}$ (see Figure 2). Thus for our problem of a linear second order differential equation with Dirichlet boundaries, this is clearly the most advan-

tageous algorithm.

However, while the specialised algorithm is good at solving eq. 2, it is, as our naming suggests, not very general. There exists a wide variety of problems that can be expressed in the form of our matrix equation, eq. 4, where the matrix A takes different forms. What we have called the general algorithm, which is almost as efficient as the specialised algorithm (see Table 1), can be used if the matrix A is tridiagonal.

If matrix A is not on a tridiagonal form neither our specialised nor our general algorithm can be used. In the latter case, the LU-decomposition can be useful. This comes at cost of a drastically increase in CPU run time as n increases.

As we can see from Figure 2, there seems to be a lower limit to the error in the specialised algorithm. From n=10 to n=10^5 the error decreases with a slope of about -2. From eq. 3 we have that the mathematical error in our approximation goes as h^2, or n^{-2}. So our error decreases as expected from the mathematical model.

From n=10^5 and up the slope increases, and from n=10^6 the error is increasing as n increases. This increase in the relative error is due to round-off errors caused by the computer.

References

[1] Pavel Okunev and Charles R Johnson. Necessary and sufficient conditions for existence of the lu factorization of an arbitrary matrix. *arXiv preprint math/0506382*, 2005.

A

Derivation of Gaussian elimination algorithm terms

Considering our generalised matrix

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_1 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_2 & b_3 & c_3 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \ddots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & 0 & a_{n-1} & b_n \end{bmatrix},$$

we will obtain our algorithm from row reduction by subtracting the first row times $\frac{a_1}{b_1}$ from the second row (I $- \frac{a_1}{b_1}$II)), and we see that the new diagonal element is given by

$$\widetilde{b_2} \equiv b_2 - \frac{a_1}{b_1} c_1.$$

Considering eq. 4 then, the corresponding change on the left hand side is given by

$$\widetilde{d_2} \equiv d_2 - \frac{a_1}{b_1} d_1.$$

Performing the same operation on the next row (III $- \frac{a_2}{\widetilde{b_2}}$II), we get

$$\widetilde{b_3} \equiv b_3 - \frac{a_2}{\widetilde{b_2}} c_2, \quad \widetilde{d_3} \equiv d_3 - \frac{a_2}{\widetilde{b_2}} \widetilde{d_2}.$$

Letting $\widetilde{b_1} \equiv b_1$, $\widetilde{d_1} \equiv d_1$, and generalising, we get:

$$\widetilde{b_i} = b_i - \frac{a_{i-1}}{\widetilde{b_{i-1}}} c_{i-1}, \quad \widetilde{d_i} = d_i - \frac{a_{i-1}}{\widetilde{b_{i-1}}} \widetilde{d_{i-1}}. \tag{4.16}$$

Additional plots

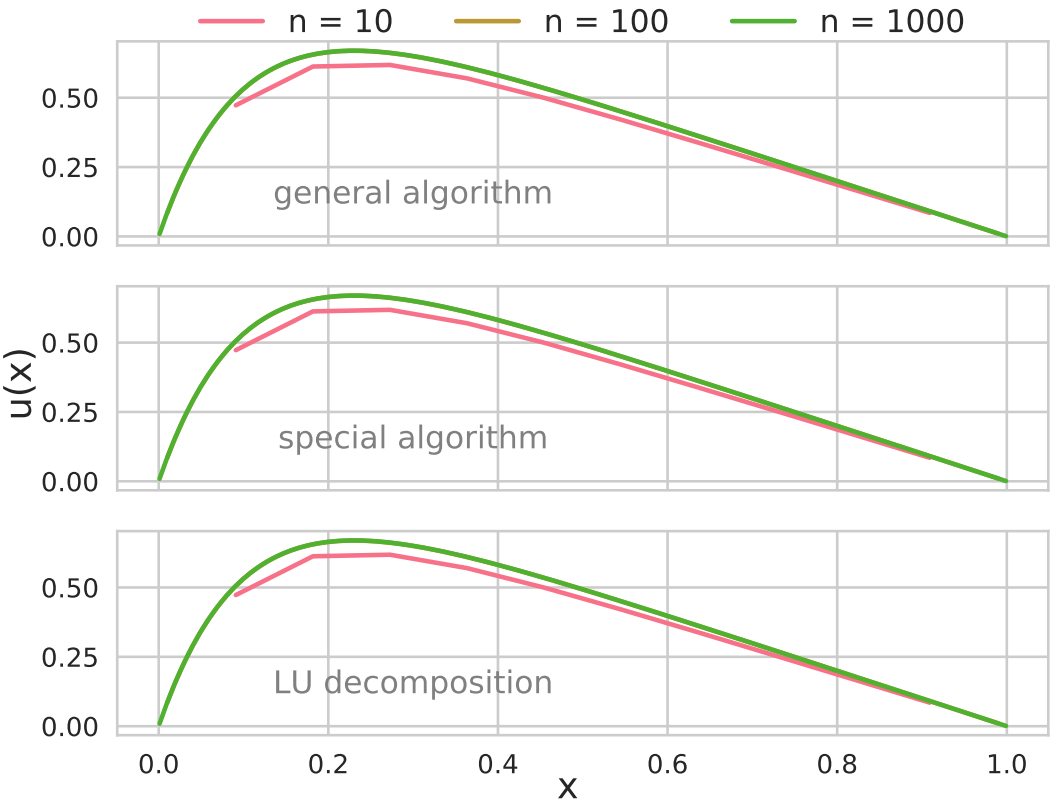

Figure 3: The numeric solution using different solving algorithms. The graphs for n=100 and n=1000 are so similar that they are not distinguishable.