

FYS4150 - Project 1: Differential Equations and Linear Algebra

Tore Klungland (torekl)

September 1, 2019

1 Introduction

Poisson's equation, given in general as

$$\nabla^2 u(\mathbf{r}) = f(\mathbf{r}) \quad (1)$$

where $f(\mathbf{r})$ is some known function, is an often-used equation in physics as it describes how a field (in this case u) is affected by some source (in this case f). If f is a complicated function this equation may not be analytically solvable, so one must use numerical methods instead, to find an approximate solution. In one dimension this can be done by using an approximate expression for the second derivative of u at discrete set of points; then equation 1 becomes a set of linear equations for the values of u at these points, which can be expressed as a matrix involving a tridiagonal matrix. Thus the problem of solving Poisson's equation numerically reduces to inverting such an equation numerically.

In this project I have examined three different methods for solving this matrix equation: One "brute-force" method which inverts the entire matrix numerically, and two which exploit the fact that the matrix is tridiagonal by storing data in vectors instead of a matrix, which reduces memory usage and speeds up the algorithm. One of these uses a general algorithm for row-reducing tridiagonal matrices, while the other takes advantage of the specific form of the matrix in question. In this text I will first outline these methods, and then compare their results, in particular the difference in CPU time used. I will also discuss the effects of round-off errors as the step size between the different data points is changed.

2 Methods

2.1 Turning the Poisson equation into a matrix equation

The target of this project was solving Poisson's equation,

$$u''(x) = -f(x), \quad (2)$$

for $f(x) = 100e^{-10x}$ for $x \in [0, 1]$ with the boundary conditions $u(0) = u(1) = 0$. This equation has an exact solution given by [1]

$$u(x) = (e^{-10} - 1)x + 1 - e^{-10x} \quad (3)$$

The first step is dividing this interval into a discrete set of points x_i for $i = 0, 1, 2, \dots, n+1$, where $x_0 = 0$ and $x_{n+1} = 1$. Thus the step size is given by $h = x_{i+1} - x_i = 1/(n+1)$. The second derivative can be approximated by [2]

$$u_i'' \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} \quad (4)$$

(obtained by manipulating the Taylor expansion of $u(x)$) where the error is of order h^2 . Here $u_i \equiv u(x_i)$, etc. Inserting this into equation 2 gives

$$-u_{i-1} + 2u_i - u_{i+1} = g_i$$

where I have defined $g_i \equiv h^2 f(x_i)$. This is a set of linear equations for u_i ; it can thus be rewritten on matrix form as (using the fact that $u_0 = u_{n+1} = 0$, by the boundary conditions):

$$\mathbf{A}\mathbf{u} = \mathbf{g} \quad (5)$$

Here A is an $n \times n$ matrix whose non-zero elements on the i 'th row are $A_{i,i-1} = -1$, $A_{i,i} = 2$, $A_{i,i+1} = -1$ (except the first row which only has $A_{0,0} = 2$ and $A_{0,1} = -1$, and the n 'th row which only has $A_{n,n-1} = -1$ and $A_{n,n} = 2$). Thus it is a tridiagonal matrix with 2 on the diagonal and -1 on the elements directly adjacent to the diagonal. Furthermore $\mathbf{u}^T \equiv [u_1, u_2, \dots, u_n]$, and $\mathbf{g}^T \equiv [g_1, g_2, \dots, g_n]$.

2.2 Row-reducing a tridiagonal matrix

The problem is now finding the reduces row echelon form of the matrix

$$[A; \mathbf{g}] = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots & g_1 \\ a_1 & b_2 & c_2 & \dots & \dots & \dots & g_2 \\ \dots & a_2 & b_3 & c_3 & \dots & \dots & g_3 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & a_{n-2} & b_{n-1} & c_{n-1} & g_{n-1} \\ \dots & \dots & \dots & \dots & a_{n-1} & b_n & g_n \end{bmatrix} \quad (6)$$

For the moment, I am not making any assumptions about the values of a_i , b_i or c_i . Renaming b_1 and g_1 to \tilde{b}_1 and \tilde{g}_1 respectively, and then subtracting $a_1/\tilde{b}_1 \times (\text{row } 1)$ from the second row leaves

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & \dots & \dots & \dots & \tilde{g}_1 \\ 0 & \tilde{b}_2 & c_2 & \dots & \dots & \dots & \tilde{g}_2 \\ \dots & a_2 & b_3 & c_3 & \dots & \dots & g_3 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & a_{n-2} & b_{n-1} & c_{n-1} & g_{n-1} \\ \dots & \dots & \dots & \dots & a_{n-1} & b_n & g_n \end{bmatrix} \quad (7)$$

where $\tilde{b}_2 = b_2 - (a_1/\tilde{b}_1)c_1$ and $\tilde{g}_2 = g_2 - (a_1/\tilde{b}_1)\tilde{g}_1$. The matrix can be brought to an upper triangular form by repeating this process, i.e

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & \dots & \dots & \dots & \tilde{g}_1 \\ 0 & \tilde{b}_2 & c_2 & \dots & \dots & \dots & \tilde{g}_2 \\ \dots & 0 & \tilde{b}_3 & c_3 & \dots & \dots & \tilde{g}_3 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & 0 & \tilde{b}_{n-1} & c_{n-1} & \tilde{g}_{n-1} \\ \dots & \dots & \dots & \dots & 0 & \tilde{b}_n & \tilde{g}_n \end{bmatrix} \quad (8)$$

where $\tilde{b}_i = b_i - (a_{i-1}/\tilde{b}_{i-1})c_{i-1}$ and $\tilde{g}_i = g_i - (a_{i-1}/\tilde{b}_{i-1})\tilde{g}_{i-1}$ for $i = 2, 3, \dots, n$. This matrix can be brought to reduced row echelon form by dividing the lower row by \tilde{b}_n (the n 'th element of the solution is then given by $u_n = \tilde{g}_n/\tilde{b}_n$), and subtracting c_{n-1} times this line from the line above it; dividing this line by \tilde{b}_{n-1} then gives the $(n-1)$ 'th element of the solution; $u_{n-1} = (\tilde{g}_{n-1} - c_{n-1}u_n)/\tilde{b}_{n-1}$. Repeating this process brings the matrix to the form

$$\begin{bmatrix} 1 & 0 & 0 & \dots & \dots & \dots & u_1 \\ 0 & 1 & 0 & \dots & \dots & \dots & u_2 \\ \dots & 0 & 1 & 0 & \dots & \dots & u_3 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & 0 & 1 & 0 & u_{n-1} \\ \dots & \dots & \dots & \dots & 0 & 1 & u_n \end{bmatrix} \quad (9)$$

where $u_i = (\tilde{g}_i - c_i u_{i+1})/\tilde{b}_i$ for $i = 1, 2, \dots, n-1$ (note that this is not the full solution as the interval is divided into $n+2$ points; however the end points u_0 and u_{n+1} are both known from the boundary conditions to be zero). Given vectors \mathbf{a} , \mathbf{b} , \mathbf{c} and \mathbf{g} , the solution \mathbf{u} can thus be found by the following two-step algorithm:

Forward substitution (general):

$$\tilde{b}_1 = b_1; \quad \tilde{g}_1 = g_1 \quad (10)$$

For $i = 2, 3, \dots, n$:

$$\tilde{b}_i = b_i - \frac{a_{i-1}}{\tilde{b}_{i-1}}c_{i-1} \quad (11)$$

$$\tilde{g}_i = g_i - \frac{a_{i-1}}{\tilde{b}_{i-1}}\tilde{g}_{i-1} \quad (12)$$

Backward substitution (general):

$$u_n = \frac{\tilde{g}_n}{\tilde{b}_n} \quad (13)$$

For $i = n-1, n-2, \dots, 1$:

$$u_i = \frac{\tilde{g}_i - c_i u_{i+1}}{\tilde{b}_i} \quad (14)$$

When programming, this requires $8n$ floating-point operations (if the factor a_{i-1}/\tilde{b}_{i-1} is calculated only once per i) (not exactly, but for large n the error is negligible). I implemented this algorithm in C++ (with $a_i = c_i = -1$, $b_i = 2$ and $g_i = h^2 f(x_i)$ with f as specified above) for $n = 10, 100, 1000, \dots, 10^7$ and wrote the solution (for $n = 10$, $n = 100$ and $n = 1000$), maximum relative error and time elapsed running the algorithm to files.

2.3 Specialization of the algorithm

In this case all $a_i = c_i = -1$, and $b_i = 2$. Thus the algorithm as described above can be specialized; first of all, $\tilde{b}_i = (i+1)/i$ (this can be proved by induction,

from $\tilde{b}_i = 2 - 1/\tilde{b}_{i-1}$). If these are pre-calculated they will not contribute to the number of floating point operations of the algorithm. This can also improve precision, as all elements are calculated directly as opposed to from repeated subtractions in the general method; for large n these repeated subtractions can lead to round-off errors building up. A specialized version of the previously described algorithm is then:

Forward substitution (specialized):

$$\tilde{b}_i = \frac{i+1}{i} \text{ for } i = 1, 2, \dots, n; \quad \tilde{g}_1 = g_1 \quad (15)$$

For $i = 2, 3, \dots, n$:

$$\tilde{g}_i = g_i + \frac{\tilde{g}_{i-1}}{\tilde{b}_{i-1}} \quad (16)$$

Backward substitution (specialized):

$$u_n = \frac{\tilde{g}_n}{\tilde{b}_n} \quad (17)$$

For $i = n-1, n-2, \dots, 1$:

$$u_i = \frac{\tilde{g}_i + u_{i+1}}{\tilde{b}_i} \quad (18)$$

Not counting the pre-calculated variables, this requires $4n$ floating-point operations. Naively one would then expect this algorithm to be roughly twice as fast as the general one; however this is not the case, as the number of memory reads/writes is the same for both cases. Thus this algorithm should be faster, but not quite twice as fast. I implemented this algorithm in C++ for the same parameters as the general one, and compared the results (in particular the run-time) to those from the general algorithm.

2.4 Solution by LU decomposition

For the present problem the algorithms as described above are the most efficient; by exploiting the tridiagonal nature of the matrix they only require a number of floating-point operations which is proportional to n . They also use relatively little memory; as only the non-zero elements of the matrix are stored the number of stored values is also proportional to n . However, for a more general matrix equation $A\mathbf{u} = \mathbf{g}$ more generally applicable methods must be used, such as LU decomposition. In such methods the entire $n \times n$ matrix is used, so that the number of stored values is proportional to n^2 .

In LU decomposition, the matrix A is written as a product of a lower and an upper triangular matrix, $A = LU$, where [2]

$$L_{ij} = \begin{cases} l_{ij} & \text{for } i > j \\ 1 & \text{for } i = j \\ 0 & \text{for } i < j \end{cases} \quad (19)$$

$$U_{ij} = \begin{cases} u_{ij} & \text{for } i \leq j \\ 0 & \text{for } i > j \end{cases} \quad (20)$$

This decomposition is possible if A is non-singular, i.e. $\det(A) \neq 0$. When this decomposition is obtained the matrix equation $A\mathbf{u} = \mathbf{g}$ can be solved in two steps (which are easier to solve than the original one due to the shape of the L and U matrices): First by solving $L\mathbf{y} = \mathbf{g}$ for \mathbf{y} , then solving $U\mathbf{u} = \mathbf{y}$ [2]. I used this method in C++ (using the library Armadillo[3][4] for LU decomposition and solving the equations) for $n = 10, \dots, 10^4$ (higher powers of 10 are not possible; $n = 10^5$ would require 10^{10} numbers to be stored, i.e. roughly 80GB of memory), and compared the CPU time with the previous algorithms.

ADD CALCULATION OF FLOATING-POINT OPERATIONS!!

References

- [1] Assignment text for project 1 in FYS3150/FYS4150, Department of Physics, University of Oslo, Norway
- [2] Hjorth-Jensen, Morten, 2015, Computational Physics - lecture notes
- [3] Sanderson, Conrad and Curtin, Ryan, Armadillo: a template-based C++ library for linear algebra, Journal of Open Source Software, Vol. 1, pp. 26, 2016
- [4] Sanderson, Conrad and Curtin, Ryan, Practical Sparse Matrices in C++ with Hybrid Storage and Template-Based Expression Optimisation, Mathematical and Computational Applications, Vol. 24, No. 3, 2019