FYS4150: Project 1

Mikael Toresen

September 15, 2013

Abstract

This project is primarily about matrix operations, but also about the usage of matrices in solving differential equations. A simple case of Poisson's equation in one dimension is solved here with Dirichlet boundary conditions. The second derivative is approximated with a 3-point formula, and solved for discrete evenly distributed points in a mesh by using a vectors representing a tridiagonal matrix. After performing a simplified Gaussian elimination one arrives at the solution. The efficiency of this method is then compared to that of the LU decomposition. The final part of the project involves comparing the efficiency of row-major and column-major matix order traversing.

Throughout the project i have used Armadillo, as using `lib.cpp` and `lib.h` proved more problematic than expected on Windows.

1 Discretizing the problem

The generic Poisson equation can be written as:

$$\nabla^2 u - f = 0$$

We will in this project solve it in its one-dimensional form with Dirichlet boundary conditions :

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0. \tag{1}$$

where $f(x)$, the source term is defined as:

$$f(x) = 100e^{10x} \tag{2}$$

This has a closed form solution given by:

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

We discretize this equation to a mesh with mesh points $x_i = ih$, $i \in [0, n+1]$, where $x_0 = 1$, $x_{n+1} = 1$. h is then he step size. The initial conditions then give us $v_0 = v_{n+1} = 0$, where v_i is the discretization of $u(x)$. The second derivative can be approximated by

$$\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} \tag{4}$$

Equation (1) therefore can be written as:

$$-(v_{i+1} - 2v_i + v_{i-1}) = h^2 f_i, \quad for i \in [1, n] \tag{5}$$

where $f_i = f(x_i)$.

This can be rewritten as a linear set of equations by noticing that Equation (4) can be rewritten as:

$$\begin{pmatrix} -1 & 2 & -1 \end{pmatrix} \begin{pmatrix} v_{i+1} \\ v_i \\ v_{i-1} \end{pmatrix} = h^2 f_i$$

The equations are of the form:

$$\begin{aligned}
&2v_1 - v_2 && = f_1h^2\\
&-v_1 + 2v_2 - v_3 && = f_2h^2\\
&\quad - v_2 \ + 2v_3 \ -v_4 && = f_3h^2\\
&\qquad - v_3 \ +2v_4 \quad -v_5 && = f_4h^2\\
&\qquad\qquad \ddots \quad \ddots && \vdots\\
&\qquad\qquad\quad -v_{n-1} + 2v_n && = f_nh^2
\end{aligned}$$

which in the same way can be rewritten as:

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & -1 & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & -1 & 2 & -1 \\ 0 & \cdots & \cdots & 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix} = h^2 \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} \tag{6}$$

Which is of the form $\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}$.

2 Implementing the program

2.1 The algorithms

The matrix $\mathbf{A}$ in Equation (6) is a tridiagonal matrix and can thus be implemented as only three vectors. For our specific purposes, however, it is sufficient to only use one vector $\mathbf{d}$ for the diagonal and let the lower and upper diagonals be represented by a constant. $\mathbf{v}$ and $\mathbf{b}$ will also be represented as vectors.

To solve Equation (6) we simplify it further by diagonalizing the elements using Gaussian elimination until the set of linear equations all are on the form $\tilde{a}_{ii}v_i = h^2\tilde{f}_i$, where a_{ii} is a matrix element of $\mathbf{A}$ along the diagonal and the tilde represents a modification from the original value of that element. We can achieve this by using a forward and backward substitution as follows:

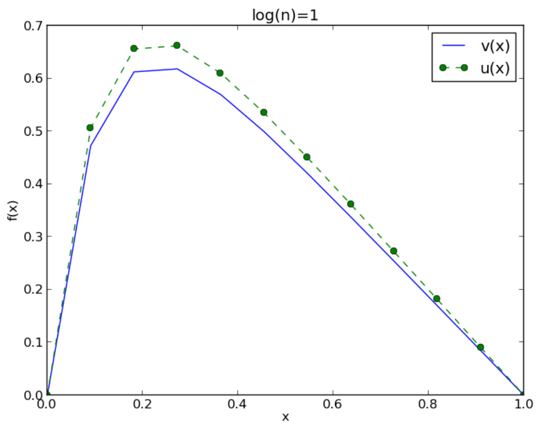
```
double ac=a*c;
for (int i=2;i<n+1;i++){
  b[i]-=ac/b[i-1];


//backward substitution
for (int i=n-1;i>0;i--)
```

Listing 1: Forward and backward substitution algorithm

where a_i, b_i and c_i are as defined in `project1_2013.pdf`. r_i is the right-hand side of Equation (6), namely h^2f_i.

We see from the algorithms in Listings 2.1 that the forward substitution requires 8 Flops. If we beforehand know that $a = c = -1$, then we can reduce the amount of flops by 2 to 6 flops. This means that for a $n \times n$ matrix we would have $6(n-1)$ flops in total. Extracting $\mathbf{v}$ then requires an additional n flops (in total $7n - 6$ flops). For both a standard gaussian elimination algorithm and a LU decomposition one would require $\mathcal{O}(\frac{2}{3}n^3)$. One therefore gains significant speedup by using these simple algorithms.



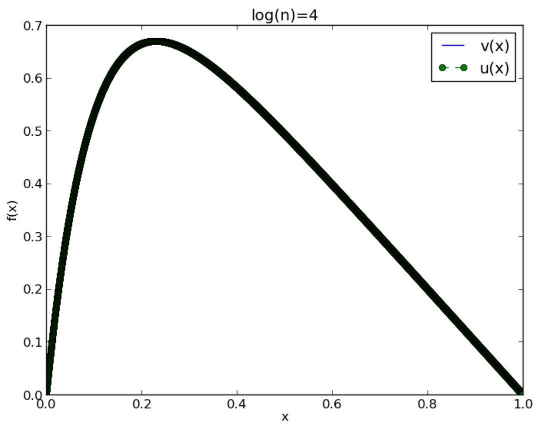


Figure 1: Plot of exact and approximated solution for $n = 10$ and $n = 10^4$. Blue line shows the numerical solution v and green circles show the exact discrete points of the solution.

2.2 Comparing solutions

Once the algorithms in Listings 2.1 are implemented one can compare the solution with the exact closed form expression by comparing $\mathbf{v}$ with $\mathbf{u}$. As we can see from Figure 1, even for only 10 points one gets a qualitatively very similar plot. This becomes even more accurate as you look at 10^4 points.

To get a better idea of the error of the computation we may look at the greatest of the relative errors ϵ_i committed in the mesh:

$$\epsilon_i = \log\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{7}$$

Where ϵ_i is the error in a point, and by using logarithmic values the errors are easier to compare. In Figure 2 we see that the error is inversely proportional with the number of points up until somewhere near $n = 10^5$. Within that range the error went down a factor of 100 for every increase in n by 10, which is equivalent to h^2 becoming approximately 100 times smaller. This shows the correspondence between h and the error. As h becomes even smaller, the small values of h and the finer differences between the numbers in the numerator of the differential become more difficult to represent.

For $n > 10^5$ the error increases because of the computer not being able to represent the numbers exact enough.

If one compares the speed of the simple algorithm used in this project with LU decomposition codes for a general matrix, one notices that while both are relatively fast for smaller n, the simple algorithm doesn't get much slower before it passes something like $n = 10^6$ points. LU decomposition however is already at $n = 10^3$ points showing strong signs of not being able to keep up. There is, in my opinion, little point in giving exact values to compare between the two as it is clear that LU decomposition will be slower when it is unable to keep the information about the matrices on cache, while the other is. This is because then the computation is not only slowed down because there are a lot of floating point operations to be done, there is also the transfer of the information to and from the processor. The latter will often dominate making it a very extreme comparison. For $n = 10^5$ the computer is extremely slow in computing the matrix, and on some computers it may run out of memory.

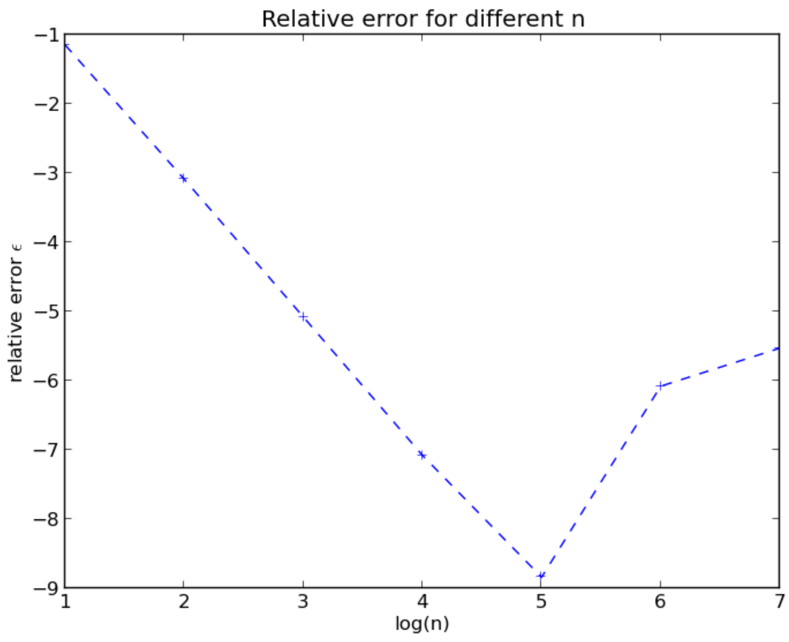


Figure 2: Plot of error ϵas function of $\log_{10} n$

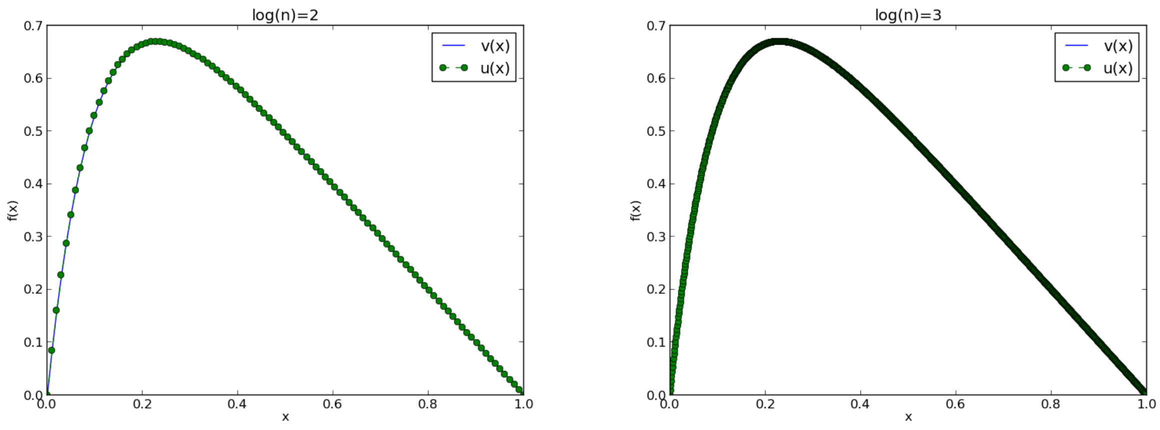


Figure 3: Plot of exact and approximated solution for $n = 10^2$ and $n = 10^3$. Blue line shows the numerical solution a and green circles show the exact discrete points of the solution.

3 Matrix operation efficiency

As part of the project we were to compare row-major with column major traversion of arrays using matrix-matrix as an example. To get more efficient row and column traversion matrices some of the indices were changed into what can be seen below:

```
for (int i=0;i<n;i++)//Row-major
  for (int k=0;k<n;k++)
    for (int j=0;j<n;j++)
      a(i,j)+=b(i,k)*c(k,j);

for (int j=0;j<n;j++)//Column major
  for (int k=0;k<n;k++)
    for (int i=0;i<n;i++)
      a(i,j)+=b(i,k)*c(k,j);
```

They were changed based on the idea that a matrix traversion would be most efficient if matrix elements that are close to eachother in memory change more often. This means that for in a row-major paradigm it would be most effective to traverse through the row before going to the next column.

Comparing these to ways to traverse a matrix to do a matrix-matrix multiplication for large n showed a clear tendency that the column major ordering was clearly most efficient. Throughout this project the vectors and matrices have been represented using the Armadillo package. The Armadillo package tries to give a *Matlab*-like syntax, and, as strongly implied by the tests that were made, a *Matlab*-like matrix traversion preference. Standard *C++* uses row-major memory allocation while Armadillo uses column-major memory allocation. This was then tested against the the BLAS matrix multiplication algorithm which is the standard non-loop traversion for Armadillo. This again was much faster, both because it uses an effective algorithm, and because it is essentially just a wrapper for FORTRAN code, which is fast.

As an example, for $n = 1000$, the row-major traversion took 16 seconds while the column major used approximately 6 seconds, which is more than twice as good. The BLAS computation was measured to take around 1 second to run. This is an order of magnitude better than the row-major traversion.

4 Implementation strength

The Poisson equation can be modelled using tridiagonal matrices. Furthermore the solution gives relatively good results with only a few points. When h is too small the computer has diffuculties representing the differential in an accurate way. The diagonalization becomes less accurate as all the diagonal elements were of order 1, but the precision needed to get the closed form expression is very high, and once the computer has to deal with both high and small numbers it can easily lead to round-off-, or even truncation errors.

The method and algorithms used work well as long as none of the diagonal elements are close to 0. If $v_i \approx 0$, then this can lead to a loss of precision for v_{i+1}. To prevent this from happening one uses *pivoting*. One changes the order of the matrix elements to minimize these problems.

Although the "vector instead of matrix"-computation lead to much more effective computation, this only works for tridiagonal matrices. Therefore, the QT-decomposition or similar methods deserve more praise than what is given in this report. For simple computations with few points it might be more effective, time-wise, to use the QT-decomposition, and thereby save time implementing the algorithm for different cases.