Solving Systems of Linear Equations with Gaussian Elimination and LU Decomposition using C++

- - - -

Project 1, FYS3150

Sunniva Jacobsen, Helén Persson and Line Gaard Pedersen

September 19, 2016

Abstract

In this report we present our studies of numerical methods for solving the Poisson equations with Dirichlet boundary conditions using C++. The equation has been solved as a set of linear equations represented as a matrix equation with an $N \times N$ tridiagonal matrix. We have mainly looked at two ways of solving the matrix equation, namely Gaussian elimination and LU-decomposition, where we compared the elapsed time for each algorithm. By comparing our numerical results with an analytical solution to a test problem, we have studied the relative error.

Our main results show that for N much greater than 1000, the Gaussian elimination was much more efficient than the LU decomposition. Reducing the number of floating point operations (FLOPS) by creating an algorithm specially designed for our matrix did not reduce the elapsed time very much. The relative error was smallest for $N = 10^5$, where it was in the order $\sim 10^{-9}$.

1 Introduction

In science, many problems can be solved numerically by using differential equations. It is therefore of great importance to find the most efficient and accurate algorithms to do so. In this paper we present our studies of solving the one-dimensional Poisson equation with Dirichlet boundary conditions using C++. Dirichlet boundary conditions simply means that the boundary conditions are known to be zero.

We have rewritten the Poisson equation

$$-u''(x) = f(x) \tag{1}$$

as a set of linear equations. This set can easily be represented by a matrix equation, as discussed in section 2.1.

The specific equation we have studied is

$$\mathbf{A}\mathbf{v} = \mathbf{b} \tag{2}$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix and $\mathbf{v}$ and $\mathbf{b}$ are vectors of dimension n.

Three different algorithms for solving equation 2 are studied in this paper. In section 2.2 we present the algorithm for the general tridiagonal matrix. In section 2.2 we present our specialized algorithm. Here we study the case where the elements of each diagonal are the same.

The third algorithm used is the standard LU- decomposition from the armadillo package for C++ .

In section 3 we present our algorithms and how they have been implemented in our C++ code. To compare the efficiency and accuracy of the different algorithms we have studied run time and error estimates for each algorithm. The error estimates have been done by comparing the results from our numeric implementation to an analytic expression. All results are presented in section 4. A discussion of the algorithms are then presented in section 5.

2 Method

2.1 Matrix representation of differential equations

In this project we have solved the one-dimensional Poisson equation (equation 1) with Dirichlet boundary conditions. The Poisson equation describes an electrostatic potential as a function of a localized charge distribution.

We can use the three-point formula to solve a second order differential equation:

$$-\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = f_i, \tag{3}$$

where f_i is the derivative of v at point x_i and v_{i+1}, v_i and v_{i-1} are the values of v at points x_{i+1}, x_i and x_{i-1}.
If we describe v and f as sets of discrete points for different points x_i, we can write equation 3 as a matrix equation with $\vec{v}$ as the unknown quantity and $\vec{f}$ as the solution:

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ \vdots \\ f_n \end{bmatrix} \tag{4}$$

. When the equation is represented in this way, we can use Gaussian elimination to solve for $\vec{v}$.

2.2 Gaussian elimination

Gaussian eliminiation is an effective way of solving sets of linear equations, by reducing the matrix to an upper triangular matrix. However, for an $N \times N$-matrix, where N is large, Gaussian elimination can require many floating point operations (FLOPS). Gaussian elimination is efficient in some cases, for example when the matrix is tridiagonal, because less FLOPS are required to reduce this kind of matrix to an upper triangular matrix.
We start with a general matrix equation of the form:

$$\mathbf{A}\mathbf{v} = \mathbf{f}. \tag{5}$$

where $\mathbf{A}$ is an $N \times N$ tridiagonal matrix and $\mathbf{v}$ and $\mathbf{f}$ are N-dimensional vectors.

$$\begin{bmatrix} a_1 & b_1 & 0 & \cdots & \cdots & \cdots \\ c_1 & a_2 & b_2 & 0 & \cdots & \cdots \\ 0 & c_2 & a_3 & b_3 & 0 & \cdots \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & \cdots & \cdots & c_{n-1} & a_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ \vdots \\ f_n \end{bmatrix} \tag{6}$$

To reduce A to an upper triangular matrix, we'll start with subtracting the first row times c_1/a_1 to the second row:

$$\mathbf{A}\Big(\text{II} - \frac{c_1}{a_1}\text{I}\Big) = \begin{bmatrix} a_1 & b_1 & 0 & \cdots & \\ 0 & \tilde{a_2} & b_2 & 0 & \cdots \\ \vdots & & & & \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \vdots \end{bmatrix} \tag{7}$$

where $\tilde{a_2} = a_2 - \frac{c_1}{a_1}b_1$ and $\tilde{f_2} = f_2 - \frac{c_1}{a_1}f_1$.
The next step is to subtract the second row times $\tilde{a_2}/c_2$ to the third row:

$$\mathbf{A}\Big(\text{III} - \frac{c_2}{\tilde{a_2}}\text{II}\Big) = \begin{bmatrix} a_1 & b_1 & 0 & \cdots & \\ 0 & \tilde{a_2} & b_2 & 0 & \cdots \\ 0 & 0 & \tilde{a_3} & b_3 & \cdots \\ \vdots & & & \ddots & \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ \vdots \end{bmatrix} \tag{8}$$

where $\tilde{a_3} = a_3 - \frac{c_2}{\tilde{a_2}}b_2$ og $\tilde{f_3} = f_3 - \frac{c_2}{\tilde{a_2}}\tilde{f_2}$.

We can continue this process all the way down to the last row. We will then end up with an easy expression for v_n: $\tilde{a_n}v_n = \tilde{f_n}$, where $\tilde{a_n} = a_n - \frac{c_{n-1}}{\tilde{a_{n-1}}}b_{n-1}$

and $\tilde{f_n} = f_n - \frac{c_{n-1}}{\tilde{a_{n-1}}} f_{n-1}$.
We end up with three different expressions for solving the set of linear equations:

$$\tilde{a_i} = a_i - \frac{c_{i-1}}{a\tilde{i-1}} b_{i-1}, \tag{9}$$

$$\tilde{f_i} = f_i - \frac{c_{i-1}}{\tilde{a_{i-1}}} f_{i-1}, \tag{10}$$

and

$$u_i = \frac{1}{\tilde{a_i}}(\tilde{f_i} - c_i u_{i+1}). \tag{11}$$

So, to find $\vec{u}$, we must perform a forwards and backwards substitution where we calculate each $\tilde{a_i}$ and $\tilde{f_i}$ in the forward substitution, and then use these to find each u_i in the backwards substitution. In total this procedure requires 9N floating point operations (6N for the forward substitution and 3N for the backwards substitution).

When we have a tridiagonal matrix with the same elements on each diagonal, we can simplify the algorithm. Since we are studying equation 4, this is similar to our case. For each row reduction, c_i and b_i stays the same. This means that the term $c_i * b_i = 1$ for each iteration. If we apply this to the algorithm we only need to do 7N floating point operations, since the term $c_i * b_i$ occurs twice.

2.3 LU-decomposition

LU-decomposition is a method that decomposes a matrix into two separate matrices: a lower triangular matrix (L) and an upper triangular (U) matrix.

This process requires more FLOPS than Gaussian elimination, but is easier to implement when using a linear algebra library, like armadillo. In total, LU-decomposition requires approximately N^3 floating point operations.
So, even though this method is easier to implement it requires a lot more time and memory (RAM).

2.4 Computing relative error

The relative error between each element in a computed data set v_i and the exact solution u_i can be computed as:

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right). \tag{12}$$

3 Implementation

All programs and figures is in the github repository `https://github.com/linegpe/FYS3150/tree/master/Project1`. In the program we have used a slightly different notation than in this report. In the program we have a_i on the lower diagonal, b_i on the main diagonal and c_i on the upper diagonal.

We have solved the Poisson equation (equation 1) with Dirichlet boundary conditions. We have benchmarked our algorithm to an equation $u''(x) = 100\exp^{-10x}$, which has the analytical solution: $u(x) = 1 - (1 - \exp^{-10})x - \exp^{-10x}$.

We have an if-test that checks if N is larger than 1500. If this statement is true, the LU-decomposition will not run, since this method requires a lot of memory and can cause the computer to crash.

```
// Check if n is too big for LU-decomposition
    int n_max = 15000; // Biggest value for n that does not crash my
         computer
    int N = 0;
    if (n >= n_max){
        N = 0;
        cout << "n-value too big to perform LU-decomposition. " << endl;
             // Aborts
    }
    else{
        N = n;
    }
```

4 Results

The first results presented in this reports shows how the time used by the different algorithms vary as N increases. Figure 1 shows a graphical view of the development in time, and each data point can be traced back to table 1. As expected there is not much difference between the special and general algorithm for all values of N.
After a given value of N, the LU-decomposition uses too much time for it to be an efficient algorithm. This is caused by the large number of FLOPS this method requires.

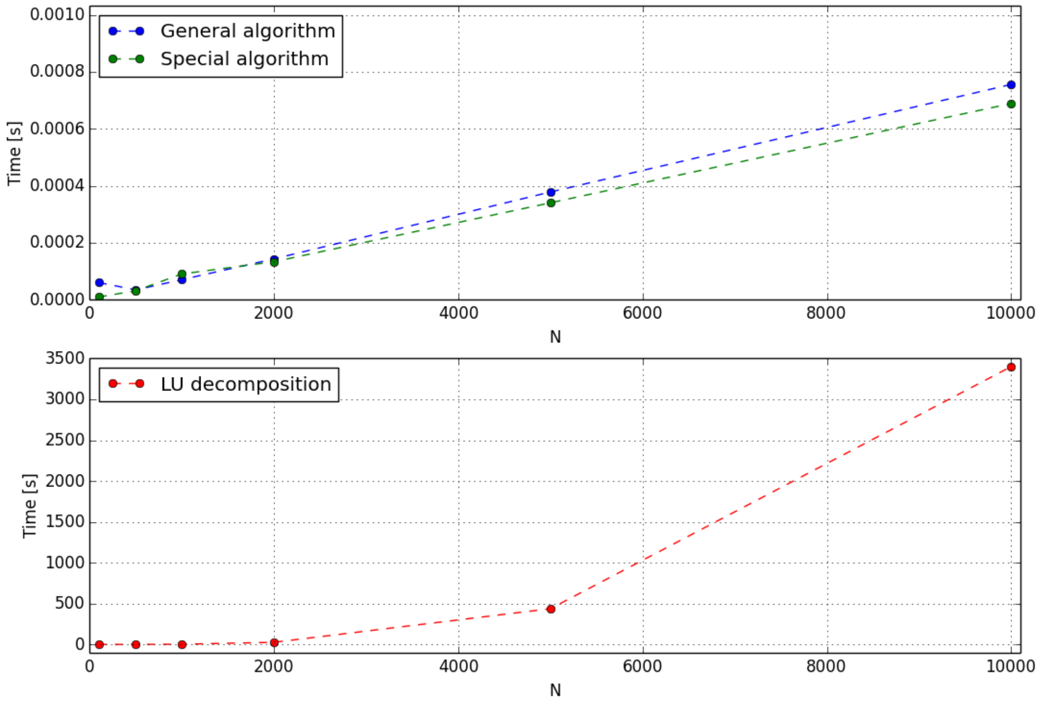


Figure 1: The figure shows the time used by the different algorithms.

Table 1: The table shows the time in seconds used by the different algorithm for different values of N.

N	General algorithm	Special algorithm	LU-decomposition
100	$6 \cdot 10^{-5}$	$9 \cdot 10^{-6}$	0.0049
500	$3.4 \cdot 10^{-5}$	$3.1 \cdot 10^{-5}$	0.341752
1000	$7 \cdot 10^{-5}$	$9 \cdot 10^{-5}$	2.78691
2000	0.000143	0.000133	26.023
5000	0.000378	0.00034	437.288
10000	0.000756	0.000689	3400.77

As seen in the following results the error in the general algorithm decreases as N increases up to $N = 10^5$. This is the critical point where round-off errors become so large that they dominate, and makes the algorithm useless for high accuracy purposes.

Table 2: The table shows ϵ (as described in equation 12, section 2.4) for the general algorithm for different values of N

N	ϵ
10	-2.33482
100	- 7.06964
500	-10.305
1000	-11.6933
2000	-13.0805
5000	-14.9137
10000	-16.3003
10^5	-20.7334
10^6	-15.5917
10^7	-13.7288

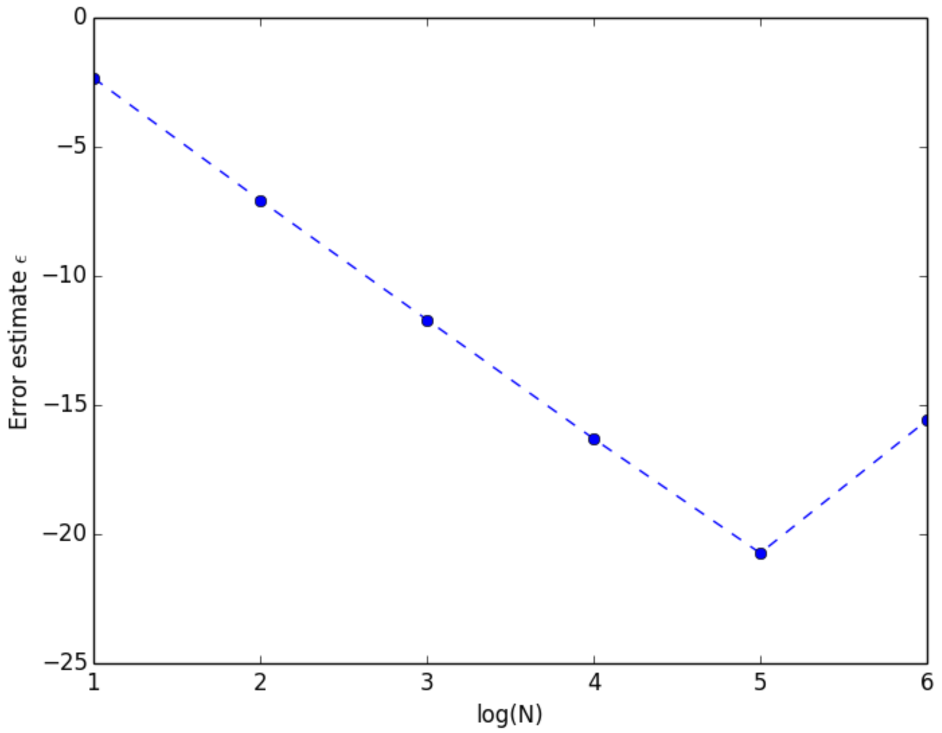
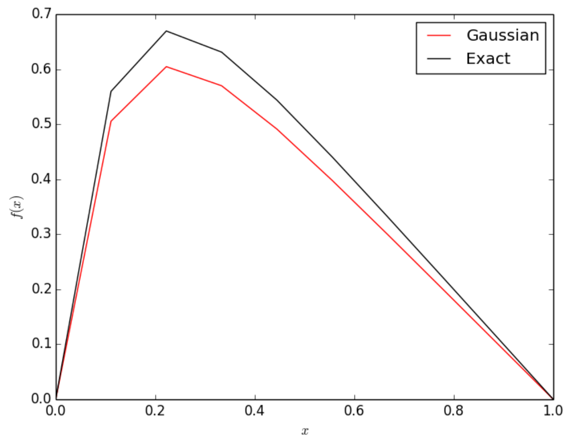
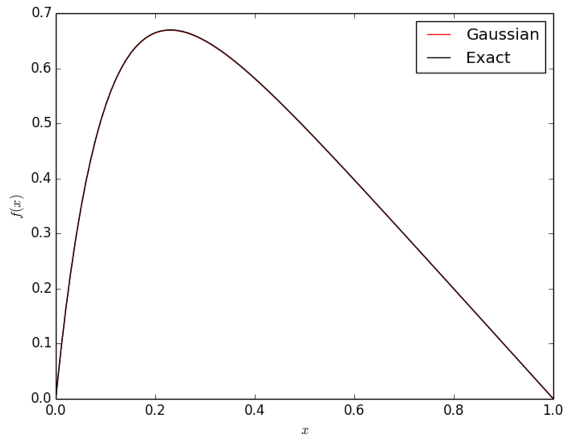


Figure 2: Plot of the development of the error in the general algorithm as N increases. Note that not all data points for the table 2 is included in the figure. Only the points contributing to interesting development in the error is included to make the plot clearer.

To verify our intuition that a larger value of N (up to $N = 10^5$) gives a better accuracy for the general method against the exact solution, we plotted the algorithm against the exact solution for two different N-values.



(a) This figure shows our Gaussian numerical solution plotted against the exact already known solution for N = 10.



(b) This figure shows our Gaussian numerical solution plotted against the exact already known solution for N = 100.

5 Discussion

In the first part of this project we implemented a general and special algorithm for solving equation 1. We compared the CPU time and number of FLOPS for these two algorithms. As we can see in figure 1 there is some difference in CPU-time, but they both seem to behave linearly as a function of the number of iterations, N. Both of these algorithms are quite fast, since the number of FLOPS is of order N (9N for the general case, and 7N for the special case). For the N-values we have studied (N up to 10^4), both of these algorithms use under 1 second to run, so the difference is not that big. For very large N, however, the time difference might be more significant.

For the LU-decomposition the CPU-time seemed to increase exponentially with N. For N $<$ 1000, LU-decomposition only took a couple of seconds to run, so not much more than the Gaussian elimination. However, for N = 5000 it took close to an hour to run! We expected the LU-decomposition to use more time than the Gaussian elimination for large N, since LU requires approximately N^3 FLOPS.

We computed the relative error for Gaussian elimination. We compared the computed solution, using both of these methods, to an analytical solution. The benchmarks in table 2 show the computed relative errors as a function of N. The relative error has a minimum for $N = 10^5$. When $N > 10^5$, the step length, h, becomes so small that round-off errors will have a significant effect. The minimum error is of order 10^{-9}. So, if higher precision is necessary, none of our algorithms are suitable.

By comparing these methods, we have found that Gaussian elimination is much more efficient than LU-decomposition. However, we only tested the case where the matrix is tridiagonal. For a dense matrix, Gaussian elimination requires more FLOPS so we cannot say that Gaussian elimination is *generally* more efficient than LU-decomposition.

Lu-decomposition is easier to implement when using a library like armadillo, so it may seem like an easy way out. However, as we have seen, this method is very limited in regard to matrix size because it requires a large amount of FLOPS. For instance, we could not run this algorithm for any matrices larger than $10^5 \times 10^5$, since this would cause a regular laptop to crash. The LU-decompositon would then require more RAM than most private computers have available.

6 Conclusion

Our aim in this report has been to evaluate a numeric algorithm as a solution to the one-dimensional Poisson equation with Dirichlet boundary conditions. We used a matrix representation of differential equations together with Gaussian elimination and LU-decomposition to calculate our solution and compare it with an exact solution. The number of floating point operations and CPU time for different methods are also briefly discussed.

As discussed in section 5 Discussion, the error in the general algorithm decreases when N increases until we hit a critical N-value of $N = 10^5$. After this point round-off errors become so large that they make the solution computed by the algorithm very uncertain.

For the CPU-time, our knowledge about FLOPS and their impact on CPU-time was proven correct. For all tested values of N, our special and general algorithm are very fast. The general algorithm is a bit slower, but makes up for this by the fact that it can handle any tridiagonal matrices. LU-decomposition is the slowest of the methods, with the largest number of FLOPS, but also the only method that can handle any type of matrix. From our results we therefore draw the conclusion that you need a computer with a very large memory, and fast computational properties for LU-decomposition to be a useful tool for large matrices.

We are pleased with our results and programs, and feel that there is not many improvements to be done on our special algorithm, specifically designed for our problem. One aspect that we would like to work on, to improve our program even further, is to automatize more of our code to make it more user friendly, and easier to plot.

7 References

1. Conrad Sanderson and Ryan Curtin. Armadillo: a template-based C++ library for linear algebra. Journal of Open Source Software, Vol. 1, pp. 26, 2016.

2. Hjorth-Jensen, Morten. Computational physics, Lecture Notes Fall 2015. August 2015.

3. Linear Algebra and its Applications: Lay, Steven R. - McDonald, Judi J. - Judi J. McDonald - Lay, David C. - David C. Lay - Steven R. Lay (2015

8 Appendix

The relative error: The reason we can use one value to represent the relative error for all points for a given N-value, except the end-points, is because our method is constructed in such a way that the error is the same in all the inner points we calculate. Figure 4 shows an example of the error in all points for a given N-value.

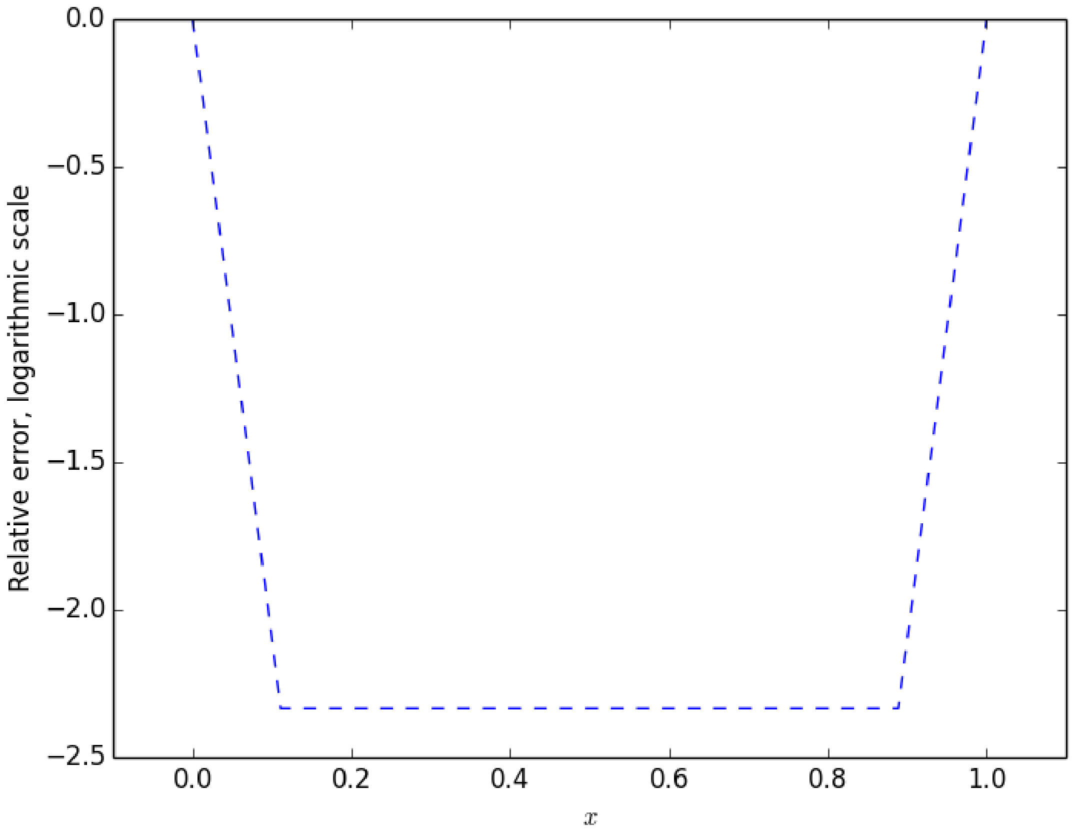


Figure 4: The relative error for a given N-value, for all computed points, x.