FYS3150 – Computational Physics
Project 1

Harald Moholt

September 2016

GitHub repository with code can be found at: github.com/harmoh/FYS3150_Project_1

1 Introduction

This project studies Gaussian elimination and LU decomposition of matrices. We will solve the one dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations. The solution is defined numerically in the interval $x \in [0, 1]$, where the step length is defined by

$$h = \frac{1}{n+1} \tag{1}$$

A linear set of equations on the form

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

will be solved, where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix and

$$\tilde{b_i} = h^2 \cdot f_i \tag{3}$$

where $f_i = f(x_i)$, and

$$f(x) = 100e^{-10x} \tag{4}$$

This problem has a solution given by

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{5}$$

An algorithm is developed for solving matrices of the sizes 10×10, 100×100 and 1000×1000, indicating iteration points of $n = 10$, $n = 100$ and $n = 1000$. The numerical solution is compared to the analytical solution and relative error is calculated. CPU time for the algorithms are measured for different n-values up to $n = 10^6$. Finally, LU decomposition is performed and compared to the algorithm in terms of relative error and time usage.

2 Method

Two different methods are used for calculating a numerical solution. The first is derived using Gaussian elimination and the second uses functions from a C++ library called Armadillo.

2.1 Gaussian elimination of a tridiagonal matrix

In order to find an algorithm to solve eq. (2), we have to be more general and start by setting up the equation.

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_1 & b_2 & c_2 & & & 0 \\ 0 & a_2 & b_3 & & & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & & & & b_{n-1} & c_{n-1} \\ 0 & 0 & \dots & \dots & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b_1} \\ \tilde{b_2} \\ \tilde{b_3} \\ \vdots \\ \tilde{b}_{n-1} \\ \tilde{b_n} \end{bmatrix} \tag{6}$$

To simplify, it can be written as a 4×4 matrix:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} \tilde{b_1} \\ \tilde{b_2} \\ \tilde{b_3} \\ \tilde{b_4} \end{bmatrix} \tag{7}$$

When performing Gaussian elimination for a 4×4 tridiagonal matrix, we start with row reduction by forward substitution for the first row, and uses $\tilde{d_1} = b_1$:

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & \tilde{b_1} \\ a_1 & b_2 & c_2 & 0 & \tilde{b_2} \\ 0 & a_2 & b_3 & c_3 & \tilde{b_3} \\ 0 & 0 & a_3 & b_4 & \tilde{b_4} \end{array}\right] = \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d_1} & 0 & 0 & \tilde{b_1}/\tilde{d_1} \\ a_1 & b_2 & c_2 & 0 & \tilde{b_2} \\ 0 & a_2 & b_3 & c_3 & \tilde{b_3} \\ 0 & 0 & a_3 & b_4 & \tilde{b_4} \end{array}\right] \tag{8}$$

For simplicity, we substitute on the right side so that $\tilde{v_1} = \tilde{b_1}/b_1 = \tilde{b_1}/\tilde{d_1}$. For the next line, we create $\tilde{d_2} = b_2 - \frac{c_1}{b_1} a_1 = b_2 - \frac{c_1}{\tilde{d_1}} a_1$ so that the next row becomes:

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d_1} & 0 & 0 & \tilde{v_1} \\ 0 & b_2 - \frac{c_1}{\tilde{d_1}} a_1 & c_2 & 0 & \tilde{b_2} - \frac{\tilde{b_1}}{b_1} a_1 \\ 0 & a_2 & b_3 & c_3 & \tilde{b_3} \\ 0 & 0 & a_3 & b_4 & \tilde{b_4} \end{array}\right] = \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d_1} & 0 & 0 & \tilde{v_1} \\ 0 & 1 & c_2/\tilde{d_2} & 0 & (\tilde{b_2} - \tilde{v_1} a_1)/\tilde{d_2} \\ 0 & a_2 & b_3 & c_3 & \tilde{b_3} \\ 0 & 0 & a_3 & b_4 & \tilde{b_4} \end{array}\right] \tag{9}$$

We substitute on the right side so that $\tilde{v_2} = (\tilde{b_2} - \frac{c_1}{\tilde{d_1}} a_1)/\tilde{d_2}$. At this point, a trend i starting to appear and we skip to the finished matrix after the forward substitution.

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d_1} & 0 & 0 & \tilde{v_1} \\ 0 & 1 & c_2/\tilde{d_2} & 0 & \tilde{v_2} \\ 0 & 0 & 1 & c_3/\tilde{d_3} & \tilde{v_3} \\ 0 & 0 & 0 & 1 & \tilde{v_4} \end{array}\right] \tag{10}$$

The trend resulted in the following formulas:

$$\tilde{d_i} = b_i - \frac{c_{i-1}}{\tilde{d}_{i-1}} a_{i-1} \qquad \text{and} \qquad \tilde{v_i} = (b_i - \tilde{v}_{i-1} a_{i-1})/\tilde{d_i} \tag{11}$$

for $i \in \{2, 3, \dots, n\}$, where $\tilde{d_1} = b_1$ and $\tilde{v_1} = \frac{\tilde{b_1}}{\tilde{d_1}}$. The implementation of the forward substitution is shown in the code below.

```
    double *d_tilde = new double[n+1];
    d_tilde[1] = b[1];
    v[1] = b_tilde[1] / d_tilde[1];
    for(int i = 2; i < n + 1; i++)
    {
        // Temporary diagonal element
        d_tilde[i] = b[i] - a[i-1] * c[i-1] / d_tilde[i-1];
        // Updating right hand side of matrix equation
        v[i] = (b_tilde[i] - v[i-1] * a[i-1]) / d_tilde[i];
    }
```

The next step is backward substitution:

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & 0 & 1 & c_3/\tilde{d}_3 & \tilde{v}_3 \\ 0 & 0 & 0 & 1 & \tilde{v}_4 \end{array}\right] = \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & 0 & 1 & 0 & \tilde{v}_3 - \frac{c_3}{\tilde{d}_3}\tilde{v}_4 \\ 0 & 0 & 0 & 1 & \tilde{v}_4 \end{array}\right] \tag{12}$$

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & 0 & 0 & \tilde{v}_2 - \frac{c_2}{\tilde{d}_2}\tilde{v}_3 \\ 0 & 0 & 1 & 0 & \tilde{v}_3 - \frac{c_3}{\tilde{d}_3}\tilde{v}_4 \\ 0 & 0 & 0 & 1 & \tilde{v}_4 \end{array}\right] = \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & \tilde{v}_1 - \frac{c_1}{\tilde{d}_1}\tilde{v}_2 \\ 0 & 1 & 0 & 0 & \tilde{v}_2 - \frac{c_2}{\tilde{d}_2}\tilde{v}_3 \\ 0 & 0 & 1 & 0 & \tilde{v}_3 - \frac{c_3}{\tilde{d}_3}\tilde{v}_4 \\ 0 & 0 & 0 & 1 & \tilde{v}_4 \end{array}\right] \tag{13}$$

Now we also see a trend for the elements on the right side with an updated v_i:

$$v_i = \tilde{v}_i - \frac{c_i}{\tilde{d}_i}\tilde{v}_{i+1} \tag{14}$$

for $i \in \{n-1, n-2, \dots, 1\}$. This formula is implemented in C++ as follows:

```
    for(int i = n - 1; i > 0; i--)
    {
        v[i] -= c[i] * v[i+1] / d_tilde[i];
    }
```

A quick review of the two algorithms reveal that the number of FLOPS to be $8(n-1)$ operations. In our case, we already know the variables of $\mathbf{A}$:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & & & 0 \\ 0 & -1 & 2 & & & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & & & & 2 & -1 \\ 0 & 0 & \dots & \dots & -1 & 2 \end{bmatrix} \tag{15}$$

where $a_i = c_i = -1$ and $b_i = 2$, the formulas for forward substitution can be written as:

$$\tilde{d}_i = 2 - \frac{1}{\tilde{d}_{i-1}} \qquad \text{and} \qquad \tilde{v}_i = (2 + \tilde{v}_{i-1})/\tilde{d}_i \tag{16}$$

And for backward substitution:

$$v_i = \tilde{v}_i + \frac{\tilde{v}_{i+1}}{\tilde{d}_i} \tag{17}$$

Then the number of FLOPS can be calculated to be $6(n-1)$. This is a huge improvement compared to standard Gaussian elimination where the number of FLOPS is $\frac{2}{3}n^3$.

2.2 LU Decomposition

This can also be called L(ower) U(pper) factorization. If we consider a non-singular matrix, such as $\mathbf{A}$, it can be decomposed into matrices $\mathbf{L}$ and $\mathbf{U}$:

$$\mathbf{A} = \mathbf{LU} \tag{18}$$

When considering 4×4 matrices, $\mathbf{A} = \mathbf{LU}$ can be written as:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \tag{19}$$

This can be rewritten as $\mathbf{L(Uv)} = \tilde{\mathbf{b}}$, where $\mathbf{Ly} = \tilde{\mathbf{b}}$ and $\mathbf{Uv} = \mathbf{y}$ in order to solve the linear equations. In this project, it was solved using the Armadillo library:

```
    v = solve(A, b_tilde);
```

The number of FLOPS for LU decomposition is $\sim O(n^2)$.

3 Implementation

Implementation of the algorithms in C++. The program was written in C++ using Qt Creator as an IDE. `main.cpp` is the main program which runs the algorithms found in `tridiagonal.cpp` for method 1 and `lu_decomposition.cpp` for method 2 (each includes a header file). Input arguments for `main.cpp` are filename, the base number for n and the exponential for n. The algorithms are run in a loop in order to calculate for several n-values at the same time, e.g. for $n = 10^1$, $n = 10^2$ and $n = 10^3$ by choosing base 10 and exponent 3. The main program also includes writing to file and measuring time for each algorithm. This part of `main.cpp` is shown below:

```
    // Open file and write to file
    string error_time = outfilename;
    error_time.append(to_string(exponent));
    error_time.append("_error_time.txt");
    ofile_summary.open(error_time);
    ofile_summary << setiosflags(ios::showpoint | ios::uppercase);
    ofile_summary << "#␣N:" << setw(18) << "h:" << setw(24) <<
                     "Error:" << setw(16) << "Time␣[sec]:" <<
                     setw(28) << "Time␣(LU␣Decomp.)␣[sec]:" << endl;

    // Declare start and final time
    clock_t start, start_lu, finish, finish_lu;
    for(int i = 1; i < exponent + 1; i++)
    {
        int n = pow(base, i);
        double h = 1.0 / (n + 1.0);

        start = clock();
        double max_error = tridiagonal(outfilename, base, i);
        finish = clock();

        start_lu = clock();
        lu_decomposition(base, i);
        finish_lu = clock();

        double time_temp = (double) (finish - start)/(CLOCKS_PER_SEC);
        double time_temp_lu = (double) (finish_lu - start_lu)/(CLOCKS_PER_SEC);
        cout << "Max error for N = " << base << "e" << i << ": " << max_error <<
                " time used: " << time_temp << " sec and " << time_temp_lu <<
                " sec." << endl;

        ofile_summary << setw(0) << setprecision(8) << base << "e" << i;
        ofile_summary << setw(18) << setprecision(8) << h;
        ofile_summary << setw(18) << setprecision(8) << pow(base, max_error);
        ofile_summary << setw(24) << setprecision(8) << time_temp;
        ofile_summary << setw(24) << setprecision(8) << time_temp_lu << endl;
    }
    ofile_summary.close();
```

When calculating the relative error, it was only found for the first method (LU decomposition took too long time). The program was run up to either $n = 10^7$ or $n = 2^{23}$, where the latter was for making a better plot with more points. A Python script was used for plotting the results written out to file. The exponent of n has to be set manually in the script before plotting.

4 Result

The first method using Gaussian elimination was implemented and produced the results for the exact and the numerical solutions. Both methods produced the exact same result. The program wrote the solutions to a text file. A Python script was used to plot the results from the text files. Plots for the numerical and exact solutions for $n = 10$, $n = 100$ and $n = 1000$ are shown in figs. 1 to 3. The difference between the numerical and the exact solution is large when $n = 10$, and is

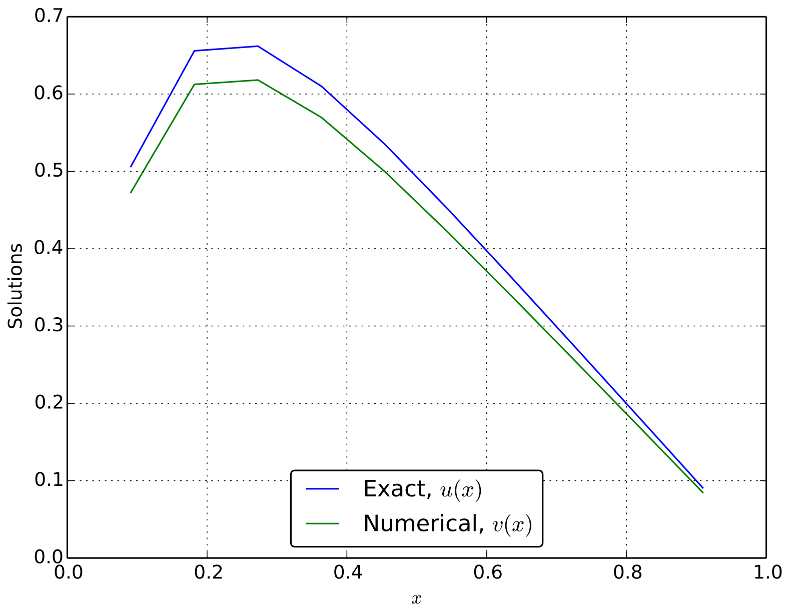

Figure 1: Exact and numerical solutions for $n = 10$

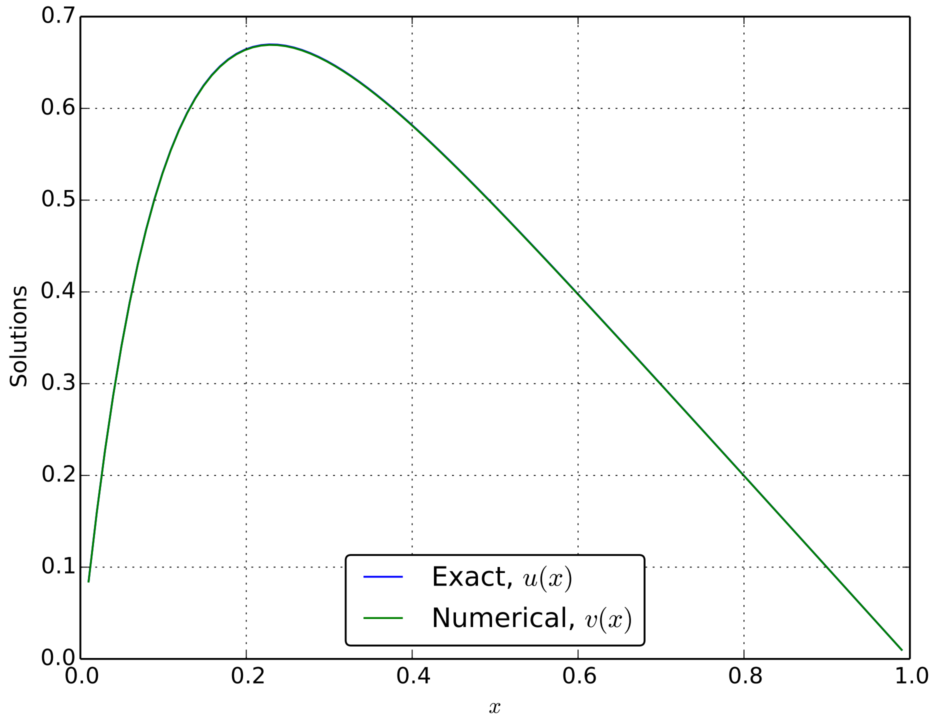

Figure 2: Exact and numerical solutions for $n = 100$

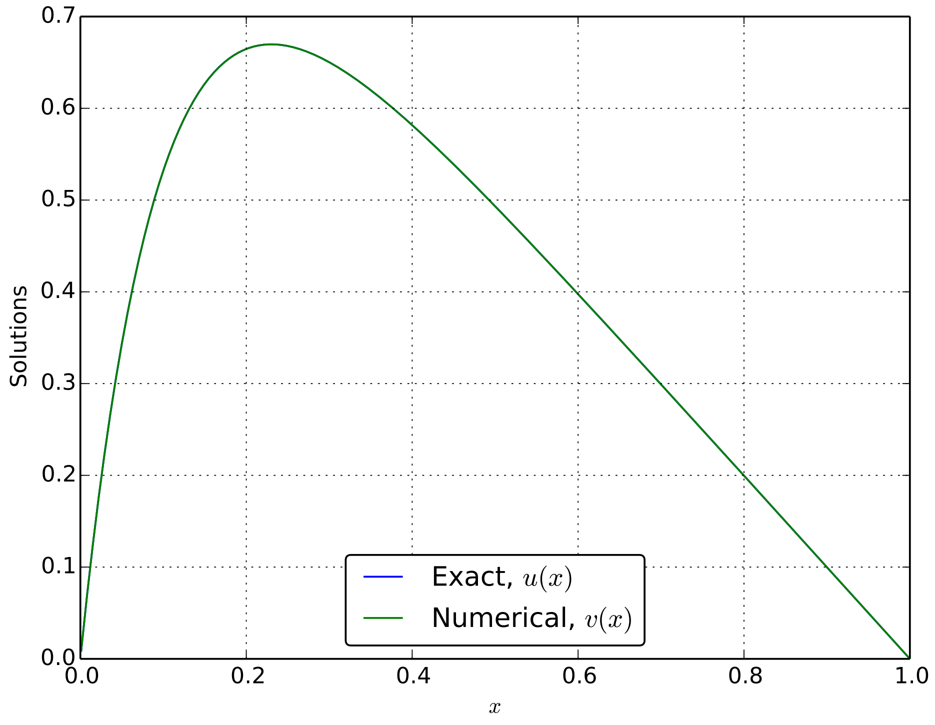

Figure 3: Exact and numerical solutions for $n = 1000$

visible in the visualization. There is no observable difference between the solutions on the other plots. One difference between the fig. 2 and fig. 3 is that when $n = 100$, the solutions are not as close the end points as when $n = 1000$. This is because of a larger step length.

Computation times for the different methods are shown in table 1. This shows the time for the first and the second method with grid points up to $n = 10^7$. LU decomposition was only used up to $n = 10^4$ due to the long computation time. The first method for the tridiagonal matrix showed a trend increasing the amount of time by approximately the same and number of grip points. The difference in computation time only become apparent when the number of grid points exceeds 10^2.

Grid points, n	Time [sec], tridiagonal method	Time [sec], LU decomp.
10e1	0.000364	0.000228
10e2	0.001154	0.001195
10e3	0.006533	0.134961
10e4	0.069644	85.563779
10e5	0.620313	-
10e6	6.010560	-
10e7	59.089966	-

Table 1: Computation time up to $n = 10^7$ compared between tridiagonal method and LU decomposition

Relative error was calculated by

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{v_i}\right|\right) \tag{20}$$

followed by a function finding the maximum relative error. Figure 4 shows a logarithmic plot of the relative error as a function of h. In the plot, n is increased by the power of 2 for a better graph. The lowest error occurs at $h \approx 10^{-5}$ where $n = 10^5$. This is where the precision error is overtaken by the round off error.

5 Conclusion

In this project, it has become clear that the number of FLOPS and computation time when solving a set of linear equations depends highly on the algorithm used. The first method was a simplified Gaussian elimination, which made it possible to perform row reduction for our specific case with only $O(8n)$ FLOPS. The second method using LU decomposition used $\sim O(n^2)$ FLOPS, which was reflected heavily in computation time. When $n > 10^3$, computation time becomes too high for any practical use. The first method did not reach this amount of computation time until $n > 10^6$. The results for each method were exactly the same. The minimum relative error was found around $n = 10^5$. With this grid size, the tridiagonal method used less than a second, while LU decomposition could not be completed due to the time consumption.

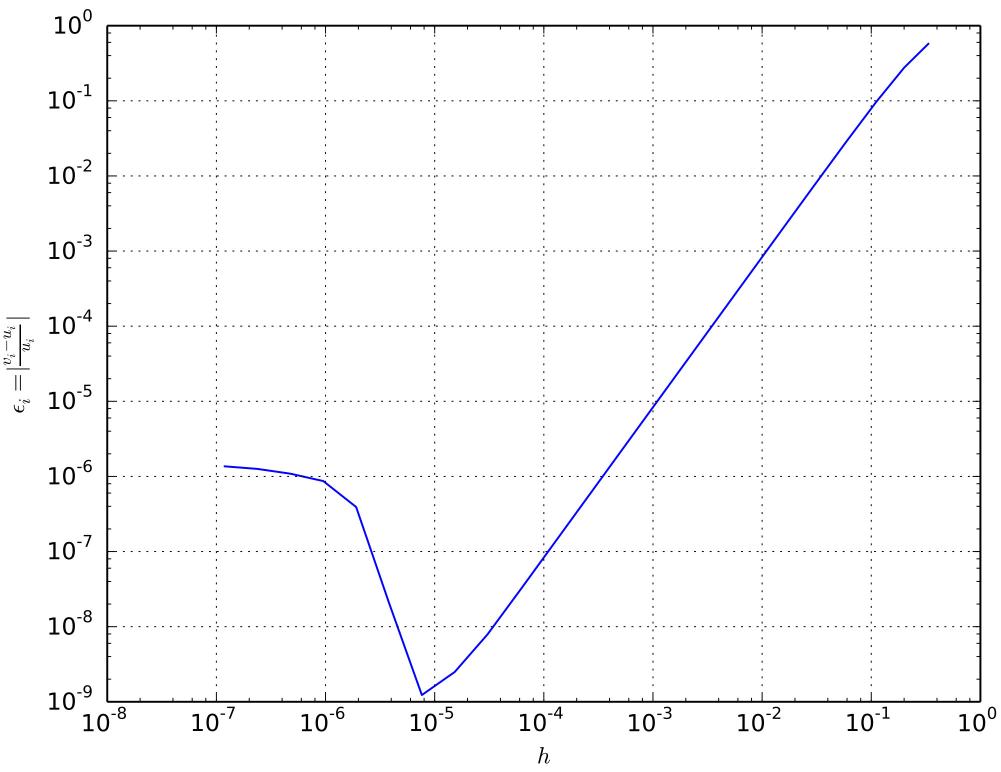

Figure 4: Relative error