UNIVERSITY OF OSLO

COMPUTATIONAL PHYSICS

Project 1

UiO : University of Oslo

Authers:

Birgitte Madsen

Soumya ??

Autumn 2015

UiO : University of Oslo

Department of Physics
University of Oslo
Sem Slands vei 24
0371 Oslo, Norway
+47 22 85 64 28
http://www.mn.uio.no/fysikk/english/

Course:

Computational Physics

Project number:

1

Hand-in deadline:

Monday 7 September, 2015

Project Members:

Birgitte Madsen
Soumya ??

Synopsis:

Copies: ??
Page count: 15
Appendices: ??
Completed: ??

PREFACE

This project is written by 6th semester physics group 4.207a at the Department of Physics and Nanotechnology at Aalborg University, Denmark, in the Spring semester, 2014, as a 10 ECTS-point bachelor project.

Reading Guide

Succeeding chapters support each other, and it is therefore recommended to read the report chronologically. When referring to equations or the like in the text, *equation* will be shortened Eq., *table* will be shortened Tab., and so forth. In App. **??** a list of frequently used symbols and constants are given. The external references used in this work appear in numbered order in brackets in the text and are listed in the bibliography at the end of the report in order of succession.

Signatures

The group member's signatures below express that the entire group is accountable for all aspects of the project and all chapters of the report.

Andreas V. Pedersen

Birgitte Madsen

TABLE OF CONTENTS

Chapter 1 Introduction **1**

Chapter 2 Method **3**
2.1 Nature of the problem 3
2.2 Description of the Algorithm 4
2.3 Source Code 6

Chapter 3 Results **7**
3.1 Reliabilty and Numerical Stability of Results 7
3.2 Interpretation of Results 7

Chapter 4 Critique **11**

Chapter 5 Conclusion **13**

Bibliography **15**

CHAPTER

1 INTRODUCTION

Hello, we love computational physics!!

$$\sqrt{(2)} = \sin(\Theta + \phi \cdot 23)$$

CHAPTER

2 METHOD

2.1 Nature of the problem

We have the one-dimensional Poisson equation with Dirichlet boundary conditions as

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0 \tag{2.1}$$

We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n \tag{2.2}$$

This equation can be rewritten as a linear set of equations of the form

$$\mathbf{Av} = \tilde{\mathbf{b}} \tag{2.3}$$

Where A is an n x n tridiagonal matrix of the form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{pmatrix} \tag{2.4}$$

To prove this first consider the matrix equation

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \ldots & \ldots & \ldots \\ a_2 & b_2 & c_2 & \ldots & \ldots & \ldots \\ & a_3 & b_3 & c_3 & \ldots & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \ldots \\ \ldots \\ \ldots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \ldots \\ \ldots \\ \ldots \\ \tilde{b}_n \end{pmatrix}. \tag{2.5}$$

Solving the above matrix equations gives a set of linear equations as

$$2v_1 - v_2 = \tilde{b}_1 \tag{2.6}$$
$$-v_1 + 2v_2 - v_3 = \tilde{b}_2 \tag{2.7}$$
$$-v_2 + 2v_3 - v_4 = \tilde{b}_3 \tag{2.8}$$
$$\vdots \tag{2.9}$$
$$-v_{n-1} + 2v_n - v_{n+1} = \tilde{b}_n \tag{2.10}$$

In general we can write it as a

$$-v_{i-1} + 2v_i - v_{i+1} = \tilde{b}_i \qquad \text{for } i = 1, \ldots, n \tag{2.11}$$

If we substitute $\tilde{b}_i = f_i h^2$ then (2.11) becomes

$$-\frac{-v_{i+1} + 2v_i - v_{i-1}}{h^2} = f_i \qquad \text{for } i = 1, \ldots, n \tag{2.12}$$

(2.12) is equal to (2.2) for second derivative of u. Thus its proved that the equation for second derivative of u can be rewritten as a set of linear equations of the form

$$\mathbf{Av} = \tilde{\mathbf{b}} \tag{2.13}$$

2.2 Description of the Algorithm

The algorithm written to solve the problem of computing the u in [1] uses the Gaussian elimination method. However, since the linear problem includes a special matrix, namely a tridiagonal matrix, the number of floating points operation needed to solve this specific problem can be meget reduced by modifying the Gaussian elimination method.

The algorithm used to solve this problem, however still, consists of the same two steps as in the Gaussian elimination: forward substitution and backward substitution.

Let us first adress the forward substitution. the aim of this first part of the algorithm is essentially to make the matrix A into an upper triangular matrix by suitable subtractions of multiplas of the first row from the other rows in the matrix. This gives rise to a change in the matrix elements. However, the elements in the first row of the matriw will not be changed, and hence we have that

$$\tilde{b}_1 = b_1 \tag{2.14}$$

in which $\tilde{b}_1$ is the element in the first column and first row of the computed triangular matrix. By writing out the computed matrix elements after subtracting multipla of the first row from the other rows to create zeros below the diagonal and using the fact that all elements right above and below the diagonal of A is equal to -1 whilst the remaining elements are 0, it is seen that the elements in the diaginal for $i > 1$, named b_i in the tridagonal matrix A and $\tilde{b}$ in the computed triangular matrix, get the value

$$\tilde{b}_i = b_i - \frac{1}{b_{i-1}} \tag{2.15}$$

[1] FiXme Note: eq-ref

Likewise, the elements in the vector $\vec{f}$ are changed to

$$\tilde{f}_i = f_i + \frac{f_{i-1}}{b_{i-1}} \tag{2.16}$$

whilst the elements named a_i in A become equal to zero, and the elements c_i are unchanged.

This gives rise to the following code for the forward substitution.

```
// Forward substitution

    double abtemp[n];
    double btemp = b[0];

    for (int i=1 ; i<n ; i++)
    {
        abtemp[i] = - 1/btemp;
        btemp = b[i] + abtemp[i];
        f[i] = f[i] - f[i-1]*abtemp[i];
        b[i] = btemp;
    }
```

[2] Notice that in the above lines of code, the first element of a vector is $i = 0$.

For every time the loop runs, there are 4 flops We have chosen to calculate $1/b_{i-1}$, which is used in both (2.15) and (2.16), to reduce the number of flopsy 1 for every time the loop is run. Since the loop runs from $i = 2$ to $i = n$, if $i = 1$ is the first element of a vector, the loop runs $n-1$ times, which gives a total number of flopsor the forward substitution of

$$\#flops = 4(n-1) \tag{2.17}$$

In the back substitution, the values of the entrances of vector $\vec{u}$ in [3] are computed. Since the result from the forward substitution is an upper triangular matrix, it is evident that

$$u_n = \frac{\tilde{f}_n}{\tilde{b}_n} \tag{2.18}$$

in which $\tilde{f}_n$ and $\tilde{b}_n$ are elements of $\vec{f}$ and $\vec{b}$ after the forward substitution. From the determined value of u_n, the values of the rest of the u_i's can be determined using the fact, that all elements in the upper triangular matrix is zero apart from the elements in the diagonal and the elements just above the diagonal, which all have the value -1. This gives

$$u_i = \frac{\tilde{f}_i + u_{i+1}}{\tilde{b}_i} \tag{2.19}$$

yielding a source code:

```
// Back substitution
```

[2] FiXme Note: skriv evt. -= osv i stedet
[3] FiXme Note: eq-ref

```
u[n-1] = f[n-1]/b[n-1];

for(int i=n-1 ; i>= 0; i--)
{
    u[i] = (f[i]-ac*u[i+1])/b[i];
}
```

For each time the loop runs, there are 2 flops Like in the forward substitution, the loop runs $n-1$ times, yielding

$$\#flops = 2(n-1)+1 \tag{2.20}$$

Hence, the total number of flopsor both the forward substitution and back substitution is

$$\#flops_{total} = 4(n-1)+2(n-1)+1 = 6n-6 \tag{2.21}$$

which gives that the number of flops goes as $6n$ or $\mathscr{O}(n)$.

[4] When comparing the number of flops using the above described algorithm to solve (2.3) in which $\mathbf{A}$ is a tridiagonal matrix and $\mathbf{v}$ is approximated by the three point formula (2.2) with the number of flops in the ordinary Gaussian elimination or LU decomposition, it is evident that this algorithm is much more efficient to solving this specific case, since the number of flops for solving a linear set of equations using the LU decomposition scales as $\mathscr{O}(n^2)$ whilst the number of flops required in the Gaussian elimination is $2n^3/3+\mathscr{O}(n^2)$. [1, 173]

2.3 Source Code

[4] FiXme Note: compare flops m. GE and LU

CHAPTER

3 RESULTS

3.1 Reliabilty and Numerical Stability of Results

3.2 Interpretation of Results

It can be shown by inserting into the (2.1) that

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{3.1}$$

is a solution to the differential equation. In the figure below ...

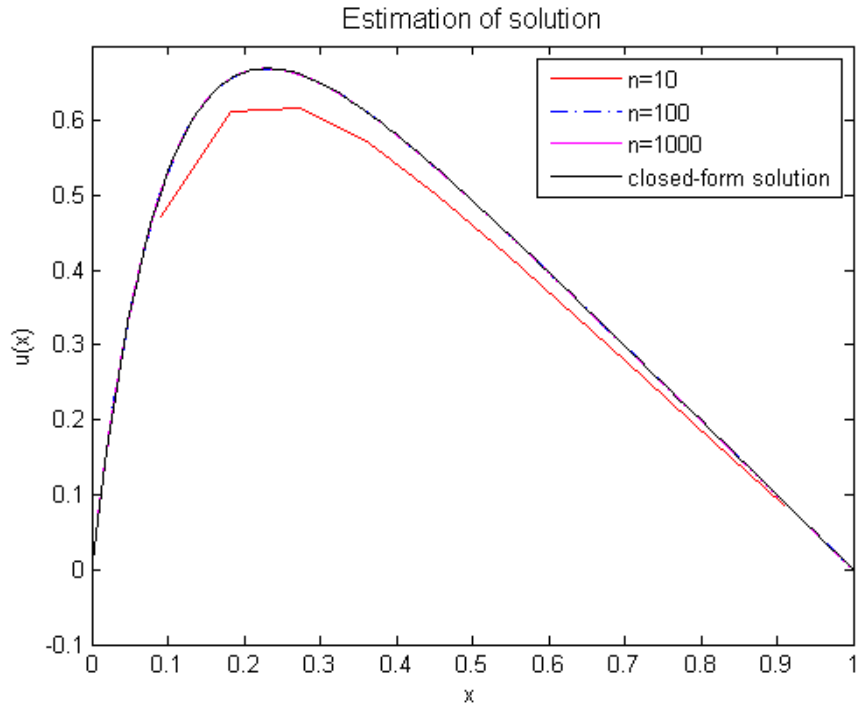

Figure 3.1. Plot of the closed-form solution together with the numerical solution for different n-values. It is evident that the precision of the numerical solution gets better when number of grid points is increased from $n = 10$ to $n = 100$ and $n = 1000$.

From Fig. 3.1 it is easily seen that by increasing the number of grid points from $n = 10$ to $n = 100$ the

precision of the solution is better. However, it is not evident from the figure that a further increment of n improves the precision further. By zooming in on the figure, it can be seen that an increment of n from 100 to 1000 actually gives a futher improvement to the numerical solution.

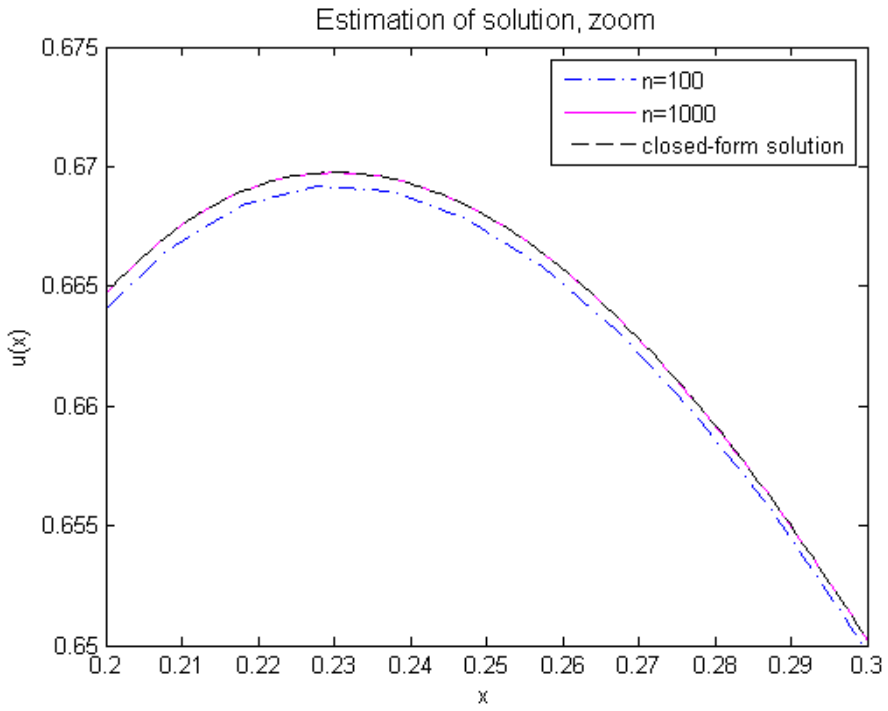

Figure 3.2. Zoom of Fig. 3.1. It is seen that a further increment of n from 100 to 1000 actually improves the precision of the result.

From Fig. 3.1 and Fig. 3.3 it is evident that there is a deviation between the closed-form solution **u** and the numerical solution gained by the algorithm made in the project, and that the deviation decreased with increasing number of grid points n. To see how this deviation actually reacts on a change in number of grid points which is directly related to the steplength, consider the relative error ε_i given by

$$\varepsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{3.2}$$

in which v_i is the i'th element of the numerical solution **v** gained by the c++ code described in Sec. 2.2, and u_i is the i'th element of the closed-form-solution **u** calculated by the formula EQ with $x = (i+1)h$ as the relation between i and x for steplength h.

n	h	$\log(h)$	ε_i
10	0.090909	-1.04139	-1.1797
100	0.009901	-2.00432	-3.08804
1000	0.000999	-3.00043	-5.08005
10000	0.000099	-4.00004	-7.07934
100000	10^{-5}	-5	-8.888

Table 3.1. The table shows different relative errors ε_i for different n's corresponding to different steplengths h. The steplength h and logarithm to the steplength is calculated in Excel, whilst the relative error ε_i is calculated as in (3.2) using the c++ code. When the number of grid points n was increased to 100,000, the precision on the fifth digit of ε_i was lost, which is why only the first four digits of ε_i for $n = 100{,}000$ are shown in the table.

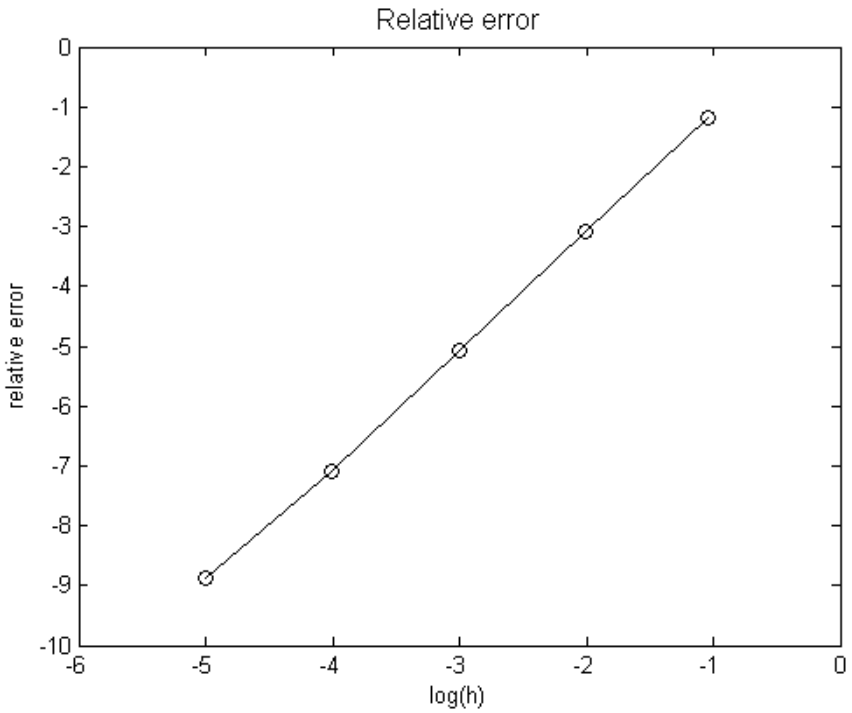

Figure 3.3. Plot of the relative error of the numerical solution to the closed-form solution as a function of the step length. These datapoints are connected by a almost straight line with a slope of approximately 2.

1

[1] FiXme Note: comment on the slope

CHAPTER

4 CRITIQUE

CHAPTER

5 CONCLUSION

BIBLIOGRAPHY

[1] M. Hjorth-Jensen, “Computational physics - lecture notes fall 2015,” August 2015.