Computational Physics - Project 1

Johannes Scheller, Vincent Noculak Lukas Powalla

September 10, 2015

Contents

1 Introduction to Project 1

In physics, we often have to deal with differential equations o second order, which can be generally written in the from

$$\frac{d^2 y}{dx12} + k^2(x) y = f(x) \quad , \tag{1}$$

where we call f the inhomgeneous term and $k^2(x)$ is a real function. A special case of these cases is Poisson's equation, which reads in the one-dimensional, spherical case

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2 \frac{d\Phi}{dr}\right) = -4\pi\rho(\mathbf{r}) \quad . \tag{2}$$

Doing some substitutions, we can write this in the following, more general form:

$$-u''(x) = f(x) \tag{3}$$

In this project, we try to solve eq.(3) with the boundary conditions $u(0) = u(1) = 0$. Therefore, we have to discretize f and u. We approximate $u(x)$ as v_i, using a grid of n gridpoints $x_i = i \cdot h$. Thus, $h = 1/(n+1)$ is our steplength. We will also write $f_i = f(x_i) = f(hi)$.
Approximating the second derivative of u, we get

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

Our goal is to solve this equation (4). Therefore, we will rewrite it as a set of linear equations in matrix form.

1.1 Rewriting the equation in matrix-form

Eq (4) can be written as a set of linear equations in matrix form. Therefore, we have to do the following steps:

$$\begin{aligned} -\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} &= f_i \\ -(v_{i+1} + v_{i-1} - 2v_i) &= h^2 \cdot f_i \qquad (5) \\ & \qquad (6) \end{aligned}$$

Assuming we have an $n \times n$-matrix $\mathbf{A}$ of the following form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix} \quad , \tag{7}$$

then we can rewrite this matrix in index notation as

$$a_{ij} = \begin{cases} 2 & \text{if } i = j \\ -1 & \text{if } |i-j| = 1 \\ 0 & \text{else} \end{cases} \quad . \tag{8}$$

Using this, we can also rewrite the multiplication $\mathbf{w} = \mathbf{A} \cdot \mathbf{v}$ with an n-dimensional vector $\mathbf{v}$ in the following way:

$$w_i \underset{\text{def}}{=} \sum_{j=1}^{n} a_{ij} \cdot v_j \underset{(8)}{=} -v_{i-1} + 2v_i - v_{i+1} \tag{9}$$

By using this result and substituting $h^2 \cdot f_i \rightarrow \bar{b}$ in eq (5), we get to the following equation:

$$\mathbf{A} \cdot \mathbf{v} = \bar{\mathbf{b}} \tag{10}$$

with matrix $\mathbf{A}$ as given above. This is a set of linear equations that we are going to solve with our programm. In this example, we will assume that $f(x)$ is given by $f(x) = 100\mathrm{e}^{-10x}$. Thus, the analytical solution of eq (3) is given by $u(x) = 1 - (1 - \mathrm{e}^{-10})x - \mathrm{e}^{-10x}$. We will later compare our numerical results to this solution.

2 Different algorithm to solve this set of linear equations

In this chapter, We want to solve the given set of linear equations first by solving it with a self-programmed algorhithm. Furthermore, we want to compare the produced numerical solution to the exact solution and calculate the maximum relative error for a given number of Gridpoints. In order to know, whether our algorithm is best to solve this set of linear equations, we want to compare this algorhith with LU-decomposition and Gaussian algorhitm with respect to the floating point operations and the calculation time.

2.1 self-programmed algorithm

As we have shown in the previous capture, you can rewrite the set of linear equations as product of a matrix and a vector:

$$\mathbf{A} \cdot \mathbf{v} = \bar{\mathbf{b}} \tag{11}$$

In this case, the matrix A is a tridiagonal matrix. This means, that it is possible to interpret this matrix as 3 vectors because all other components of the matrix are zero and will stay zero. This gives us the advantage that we don't have to deal with 2-dimensional Arrays and we can dump the number of floating operations (compared to LU-decomposition/Gaussian-algorithm)

The programmed algorithm will work in the following way. First, we want to reach a upper diagonal matrix by vector-additions/subtractions. Second, We want to bring the matrix to a diagonal form and at last, we scale the solution vector. The first two steps also effekt the solutionvector (in our case btilde[]). We programmed following souce-code (extract):

extract of the used algorithm((12 floating operations):

```
//first
for(int i=0;i<n+1;i++){
        b[i+1]=b[i+1]-c[i]*(a[i+1]/b[i]);
        btilde[i+1]+=-(a[i+1]/b[i])*btilde[i];
}
//second
for(int i=n;i>0;i--){
        btilde[i-1]+=-btilde[i]*(c[i-1]/b[i]);
}
//normalization
for(int i=0;i<n+1;i++){
        btilde[i]=btilde[i]/b[i];
}
```

As you can see, we use at the moment 12n floating point operations for the mainalgorithm. If we look closer at the algorithm, we will see that you can simplify a lot by skipping unnecessary arithmetic operations. We managed to get to 6 Floating point operations:

```
//first
for(int i=0;i<n+1;i++){
        b[i+1]=b[i+1]-1/b[i];
        btilde[i+1]=btilde[i+1]+btilde[i]/b[i];
}
//second
for(int i=n;i>0;i--){
        btilde[i-1]=(btilde[i-1]+btilde[i])/b[i];
}
```

2.2 Gaussian elemination

2.3 LU-decomposition

3 Resulults and discussion of the self-programmed algorithm

discussion... errors...plots... gerneral things about why things appear..

4 Comparison of self-programmed algorithm, Gaussian elemination and LU Decomposition