FYS3150/4150 - Project 1
Nummerical methods for solving Poisson's equation

Aram Salihi[1], Adam Niewelgowski[1]
[1]Department of Physics, University of Oslo, N-0316 Oslo, Norway

September 9, 2018

Abstract

In this article we have chosen to study three different algorithms (Thomas algorithm, a tailored verison of the thomas alogorithm and the famous LU decomposisition) for solving poission equation. We wish to compare the speed and the precision of these three methods. We found out that the LU decomposisition is siginificantly slower and less precise compared with two other algorithms. GIT repository.[1]

Contents

1 Introduction:

The Poission equation is one of the most important and widely used differential equation in different field of sciences, especially in physics (e.g arises when solving different potentials in theoretical physics).

$$\nabla^2 u = f \tag{1}$$

There is few cases in science general that this equation can be solved analytically, but quite often it is not solvable, therefore it is crucial to derive nummerical alogorithms which is time efficient

[1]`https://github.com/poiulki1/FYS4150/`

and precise with minimal error. By using dirichlet boundaries and discretizing (1) this can be turned into a simple linear algebra problem on the form:

$$A\mathbf{u} = \mathbf{f} \tag{2}$$

Where A is a $N \times N$ tridiagonal matrix. We wish to investigate three different alogorithms mentioned in the abstract to solve this linear algebra problem. This problem can easily be solved by LU decomposisition but as we later find out, this algorithm turns out be very slow when increasing the dimension of the matrix. Due to the nature of tridiagonal matrix a more time effiecient and more precise method can be derived. The Thomas algorithm is a simple method which uses foward and backward Gaussian elimination. We will first present the general Thomas algorithm, a tailored version and LU decomposisition. We will then compare the result and time the time used when running N iteration.

2 Theoretical Model:

2.1 Discretizing Possion's equation:

For the sake of simplicity we will reduce Poisson equation to a one dimensional problem on the form:

$$\frac{\mathrm{d}^2 u(x)}{\mathrm{d}x^2} = f(x)\,, \quad x \in (0,1) \tag{3}$$

By assuming that the function v is a smooth function, we will be using a taylor approximation up to fourth degree and a approximation for the second derivative can be derived:

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} \approx -\frac{2u(x) - u(x+h) - u(x-h)}{h^2} + \mathcal{O}(h^2) \tag{4}$$

(See appendix for derivation)By discretizing the function $u(x_i)$ to u_i and defining the grid points as $x_i = ih$, where h is defined to be $h = 1/(N+1)$. The above approximation can be formulated as:

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} \approx -\frac{2u_i - u_{i+1} - u_{i-1}}{h^2} + \mathcal{O}(h^2) \tag{5}$$

By imposing Dirichlet boundaries $x_0 = 0$ and $x_{n+1} = 1$ we have that:

$$u(0) = u(1) \to u_0 = u_{n+1} = 0 \tag{6}$$

Using this approximation and substituting this into Poissons equation, we will then have:

$$-\frac{2u_i - u_{i+1} - u_{i-1}}{h^2} = f_i \tag{7}$$

where $f_i = f(x_i)$ for $i = 1, ..., n$. Using the imposed conditions we can turn this problem into a simple linear algebra problem on the form:

$$A\mathbf{u} = \mathbf{f} \tag{8}$$

where $\mathbf{u} = \{u_1, ..., u_n\}$, $\tilde{\mathbf{f}} = h^2\{f_1, ..., f_n\}$ and A is a $n \times n$ tridiagonal matrix on the form:

$$A = \begin{pmatrix} 2 & -1 & 0 & . & . & 0 \\ -1 & 2 & -1 & . & . & . \\ . & \cdots & \cdots & \cdots & . & . \\ . & . & \cdots & \cdots & -1 & 0 \\ . & . & . & -1 & 2 & -1 \\ 0 & . & . & 0 & -1 & 2 \end{pmatrix} \tag{9}$$

As we see the triadiagonal matrix has 2 along its diagonal elements and -1 on the upper and lower diagonal elements and has otherwise 0 on the matrix elements.

2.2 Solving a general tridiagonal problem (deriving Thomas algorithm):

Recall the tridiagonal matrix from the previous section, we will now introduce a general $n \times n$ tridiagonal matrix on the form:

$$\underbrace{\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_1 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_2 & b_3 & c_3 & 0 & \dots \\ 0 & 0 & a_3 & b_4 & c_4 & \dots \\ \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & a_{n-1} & b_n \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_n \end{pmatrix}}_{\mathbf{u}} = \underbrace{\begin{pmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \\ \vdots \\ \tilde{f}_n \end{pmatrix}}_{\tilde{\mathbf{f}}} \tag{10}$$

In order to solve this equation we must perform a forward and a backward substitution. By performing Gaussian elimination on the second row in order to get a pivot coloumn, we must multiply the first row with a_1/b_1 and substract the first row with the second row. This will give us the matrix:

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ 0 & d_2 & c_2 & 0 & \dots & 0 \\ 0 & a_2 & b_3 & c_3 & 0 & \dots \\ 0 & 0 & a_3 & b_4 & c_4 & \dots \\ \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} \tilde{f}_1 \\ s_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \\ \vdots \\ \tilde{f}_n \end{pmatrix} \tag{11}$$

Where we have defined d_2 and s_2 to be:

$$d_2 = b_2 - c_1 \frac{a_1}{b_1} \qquad s_2 = \tilde{f}_2 - \tilde{f}_1 \frac{a_1}{b_1} \tag{12}$$

By repeating this procudere we will see a pattern, and the foward substitution is derived to be (if we define a initial conditon, $d_1 = b_1$ and $\tilde{f}_1 = s_1$):

$$d_i = b_i - c_{i-1} \frac{a_{i-1}}{d_{i-1}} \qquad s_i = \tilde{f}_i - s_{i-1} \frac{a_{i-1}}{d_{i-1}} \tag{13}$$

We have now successfully produced pivot elements on every coloumns in the matrix.

$$\begin{pmatrix} d_1 & c_1 & 0 & \dots & \dots & 0 \\ 0 & d_2 & c_2 & 0 & \dots & 0 \\ 0 & 0 & d_3 & c_3 & 0 & \dots \\ 0 & 0 & 0 & d_4 & c_4 & \dots \\ \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & 0 & d_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ \vdots \\ s_n \end{pmatrix} \tag{14}$$

The equations above can be iterated from $i = 1$ to n and will give us the correct matrix elements when its reduced to its row echelon form. In order to achieve the solution set $\mathbf{u}$ we must perform a backward substitution. Consider following (written out the matrix equation):

$$b_1 v_1 + c_1 v_2 = s_1 \tag{15}$$
$$d_2 v_2 + c_2 v_3 = s_2 \tag{16}$$
$$. \tag{17}$$
$$. \tag{18}$$
$$. \tag{19}$$
$$d_{n-1} u_{n-1} + c_{n-1} u_n = s_{n-1} \tag{20}$$
$$d_n u_n = s_n \tag{21}$$

If we look at the second last and last equation in (21) and solve for u_n and u_{n-1} we have:

$$u_n = s_n/d_n \qquad u_{n-1} = \frac{s_{n-1} - u_n c_{n-1}}{d_{n-1}} \tag{22}$$

changing the dummy index n to j and using $u_n = s_n/d_n$ as the initial value when performing backward substitution, the equation we obtain is:

$$u_{j-1} = \frac{s_{j-1} - u_j c_{j-1}}{d_{j-1}} \tag{23}$$

Where j iterates from $j = n-1$ down to $j = 1$. Thus the general algorithm to solve the general tridiagonal matrix problem is:

Foward substitution

$$d_i = b_i - c_{i-1}\frac{a_{i-1}}{d_{i-1}} \qquad 1 \leq i < N+1 \tag{24}$$

$$s_i = \tilde{f}_i - s_{i-1}\frac{a_{i-1}}{d_{i-1}} \qquad 1 \leq i < N+1 \tag{25}$$

Backward substitution

$$u_{j-1} = \frac{s_{j-1} - u_j c_{j-1}}{d_{j-1}} \qquad N-1 \leq j \leq 1 \tag{26}$$

As we later see, this method turns out be a very efficient and precise algorithm when comparing to LU-decomposisition.

2.2.1 FLOPS:

The first variable d_i execute three different operation which result in

$$FLOPS = 3(N-1) \tag{27}$$

Since the factor $\frac{a_{i-1}}{d_{i-1}}$ is in both equation (24) and (26), the number floops in 26 reduces down to

$$FLOPS = 2(N-1) \tag{28}$$

Since the $\frac{a_{i-1}}{d_{i-1}}$ can be precomputed. Lastly u_{j-1} executes three different mathematical operation, resulting

$$FLOPS = 3(N-1) \tag{29}$$

This give total $8(N-1)$ FLOPS. Since we are not taking $v_{N+1} = s_{N+1}/d_{N+1}$ into account, this reduces the total FLOPS to:

$$FLOPS = 8n - 7 \tag{30}$$

2.3 Optimizing Thomas algorithm for a special case:

It turns out that the previous algorithm which is already quite fast can be modified to a faster algorithm. Recall from previous that the diagonal element had values $a_1, ..., a_{n-1}$, $b_1, ..., b_{n-1}$ and $c_1, ..., c_{n-1}$ where we assumed that none of these values were the same to keep it general. We will now define that:

$$a_1, ..., a_{n-1} = -1 \tag{31}$$

$$b_1, ..., b_{n-1} = 2 \tag{32}$$

$$c_1, ..., c_{n-1} = -1 \tag{33}$$

By doing this we can rewrite our general algorith to fit this specific case:

Foward substitution

$$d_i = \frac{i+1}{i} \qquad s_i = \tilde{f}_i + \frac{s_{i-1}}{d_{i-1}} \qquad 1 \leq i < N+1 \tag{34}$$

The full derivation of d_i is shown in section (8)

Backward substitution

$$u_{j-i} = \frac{s_{j-1} + u_j}{d_{j-1}} \qquad N-1 \leq j \leq 1 \tag{35}$$

2.3.1 FLOPS:

Notice that d_i went from executing three operation down to two operation, but since this can be precomputed we dont need to take account for this. s_i went from three to two operation, $2(n-1)$ and u_{j-1} went from three to two operation, $2(n-1)$. Thus this simplification reduces

$$FLOPS = 8n - 7 \tag{36}$$

down to

$$toFLOPS = 4n - 4 \tag{37}$$

as previously explained since we are not taking $v_{N+1} = s_{N+1}/d_{N+1}$ into account, this reduces the total FLOPS to:

$$FLOPS = 4n - 3 \tag{38}$$

Thus making the algorithm more efficient than the general algorithm and reducing the runtime by almost a factor of 2.

2.4 LU decomposition:

LU decomposition is a effective way to solve a matrix equation. The beauty of this algorithm is that it decomposes a general matrix A, into two lower and upper triangular matrices L and U.

$$A\mathbf{v} = \tilde{\mathbf{f}} \rightarrow (LU)\mathbf{v} = \tilde{\mathbf{f}} \tag{39}$$

This is easier and more efficient way to solve a matrix equation. Since our triadiagonal matrix is invertible, instead of inverting dircectly it is more effiecient to decompose it. This saves some computational time. We can now express this as two matrix equation which is easier to solve:

$$L\mathbf{y} = \tilde{\mathbf{f}} \tag{40}$$

and:

$$U\mathbf{v} = \mathbf{y} \tag{41}$$

Where $\mathbf{v}$ is the solution set. The psuedo code for this algorithm is explained and implemented down below (3.3).

3 Method:

Every algorithm listed below is implemented in C++. Armadillo library is used in LU-decomposition.

3.1 General algorithm:

Thomas algorithm has two loops, forward substitution computing the pivot coloumns, followed by a backward substitution which finds the solution set **v** as previously explained. The C++ code is implemented down below:

```
//forward
for(int i = 2; i <= N+1; i++){
        b[i] -= (a[i-1]*c[i-1])/b[i-1];
        s[i] -= s[i-1]*(a[i-1]/b[i-1]);
    }
//backward
v[N-1] = s[N-1]/b[N-1]; //initial - last element
for(int i = N+1; i > 1; i--){
        v[i-1]= (s[i-1] - c[i-1]*v[i])/b[i-1];
    }
```

note that the algorithm allows every value of the matrix elements to be different as derived in section (2.2).

3.2 Specific algorithm:

The C++ code for the specific matrix is implemented down below.

```
//forward
    for(int i = 1; i <= N+1; i++){
        b[i] = (i+1.0)/((double)i);
        s[i] += s[i-1]/(b[i-1]);
    }
//backward
    v[N-1] = s[N-1]/b[N-1]; //initial - last element
    for(int i = N+1; i > 1; i--){
        v[i-1] = (s[i-1] + v[i])/b[i-1];
    }
```

The algorithm still needs to run through two loops with a few operations, but we clearly see the simplicity of this algorithm compared with general one.

3.3 LU-decomposition:

For calculation with LU-decomposition, we need first to implement initialization of the matrix A.

```
    for(int i = 0; i < N; i++){
        function[i] = dt*dt*f(dt*i);
        A(i,i) = b(i);
        if(i+1 != N){
            A(i+1,i) = a(i);
            A(i,i+1) = c(i);
        }
    }
```

where $a(i) = c(i) = -1$ and $b(i) = 2$.

Then we can use '*lu*' and '*solve*' module from '*armadillo*' to find the solution, this is shown in section (2.4)

```
    arma::lu(L,U,A);
    y = arma::solve(L,function,arma::solve_opts::no_approx);
    x = arma::solve(U,y,arma::solve_opts::no_approx);
    for(int j = 1; j < N+2; j++){
        v[j] = x[j-1];
    }
    v[0] = v[N+1] = 0;
```

Since the solution set **x** does not include the boundaries condition, we must then map the numerical solution **x** on the **v**. We have now the desired solution set which we can plot.

3.4 Precision and error:

To be able to say something about the accuracy of the numerical solution, we can calculate the relative error given by

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{42}$$

Where u_i is the discretized exact solution and v_i is the simulated solution. The error is calculated for multiple values of n. The error is found by taking the maximum of ϵ_i. The relative error is then plotted against the logarithm of stepsize $\log_{10} h = \log_{10}(1/(N+1))$.

4 Results:

4.1 Convergence of the numerical solution:

4.1.1 The general method (Thomas algorithm):

Below in Figure 1, we can see how fast the numerical solution approaches the analytic solution. The x-axis spans the given intervall with different number of mesh-points, in total $N+2$ points (including boundaries).

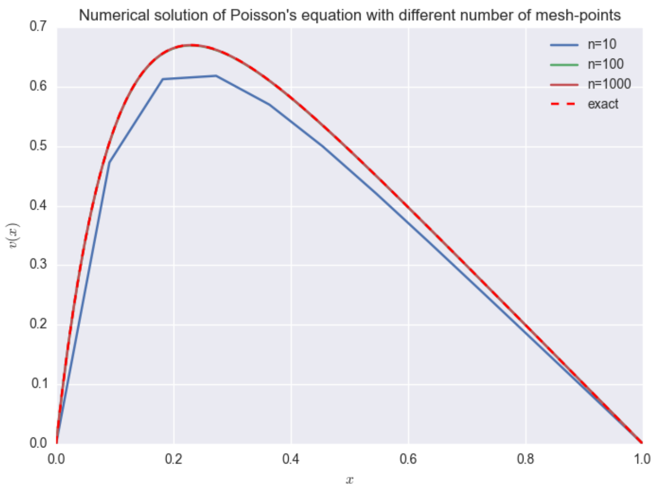

Figure 1: Plot of simulated solution compared with the exact function

4.1.2 Discussion:

The difference in the nummerical solution in the general and specific Thomas algorithm is identical. Both diagram start from below and converge upwards and almost overlap the exact solution. This is shown down below:

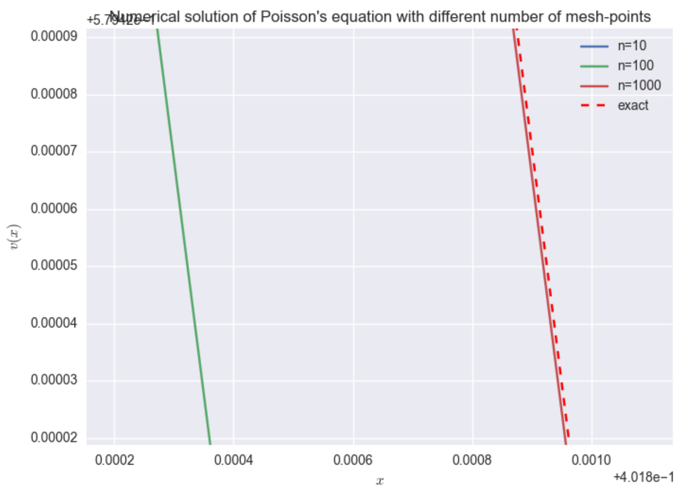

Figure 2: skrive noe

For $N = 10^3$ we do clearly see an almost overlap, but notice when increasing N more than $N = 10^5$ we get computational round off errors. See section (4.2) for further explaination.

4.1.3 LU-decomposition:

A diagram over the solution set calculated from LU compared with the exact solution.

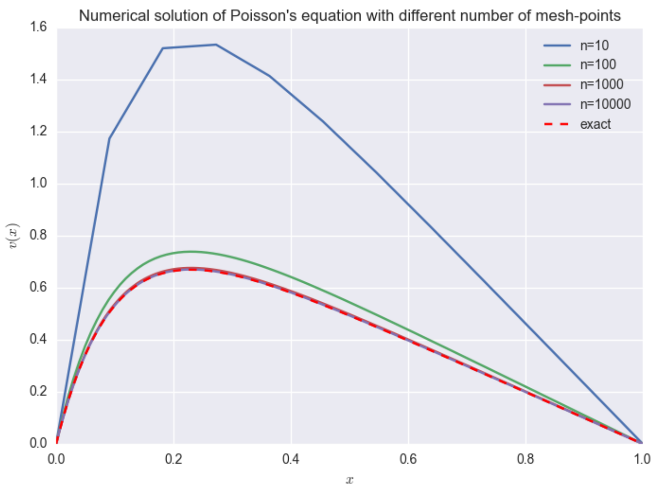

Figure 3: Plot of simulated solution produced by LU-decomposition, compared with the exact function for different number of mesh-points

4.1.4 Discussion:

In figure 2, we can see the contrast between Thomas algorithm and LU-decomposition. Instead of convering down below and upwards, it converges above and down to the exact solution.

4.2 Error:

Figure 2 shows the logarithmic plot of the relative error of the approximated solution given by equation (42).

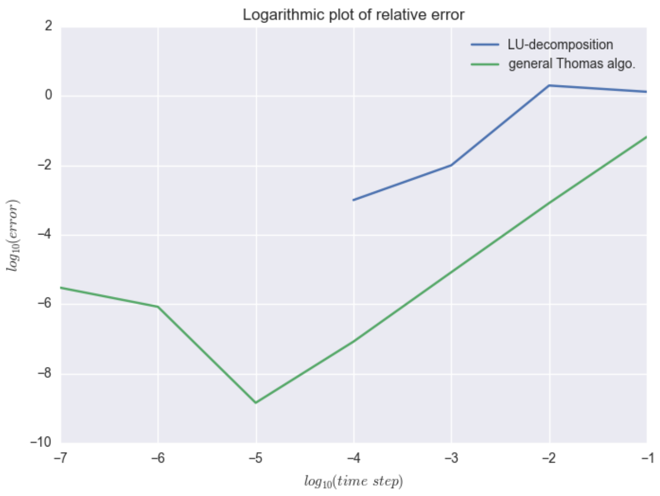

Figure 4: Logarithmic plot of the relative error. The blue graph is the LU decomposition and the green is the general Thomas algorithm.

4.2.1 Discussion:

Note the slope of the error on the interval $(-5, -1)$. Slope is exactly equal 2, which confirms the error is of the second order $\mathcal{O}(h^2)$. Note that the error in Thomas algorithm is increased when passing $N = 10^5$ mesh points. This is because of truncation error and nummerical round off error. The truncation error as we know decreases with h^2, and nummerical round off error goes as:

$$\epsilon = \frac{3\epsilon^*}{h^2} f(\xi), \qquad f\xi \in (a-h, a+h) \tag{43}$$

Where ϵ^* is the smallest distance between two floating points, ([Mørken, 2015]) Further notice that this method is also siginificantly more accurate than the LU-decomposistion when analyzing the diagram. Thomas algorithm has an error which goes with $\mathcal{O}(h^2)$ and the LU decomposition goes with $\mathcal{O}(h^3)$. See ([Hjorth-Jensen, 2015] section 6.4), not only that the speed is also different. This is presented in next section (4.3).

4.3 Speed comparison:

	Calculation time in seconds [s]		
N - number of mesh-points	Thomas - General	Thomas - Specific	LU-decomposition
10^1	3×10^{-6}	2.96×10^{-6}	1.62×10^{-4}
10^2	3×10^{-6}	2.892×10^{-6}	4.208×10^{-3}
10^3	2.4×10^{-5}	2.353×10^{-5}	1.21127×10^{-1}
10^4	3.39×10^{-4}	1.87×10^{-4}	47.9372
10^5	2.623×10^{-3}	2.047×10^{-3}	N/A
10^6	2.3623×10^{-2}	2.113×10^{-2}	N/A
10^7	2.51272×10^{-1}	1.9968×10^{-1}	N/A

4.3.1 Discussion:

As we analyze the speed, we clearly see that the specific method is just a bit faster than the general Thomas algorithm. Up to $N = 10^3$ the difference in time spent the algorithm speed is quite similiar. After $N = 10^3$, the spefic method is the fastest. Notice that the LU decomposition method is the slowest. Comparing this result with the error diagram in previous section, we clearly see that this method is less precise since the error goes as $\mathcal{O}(N^3)$ and Thomas algorithm as $\mathcal{O}(N^2)$. Further notice that the LU-decomposisition fails after $N = 10^4$ due to lack of memory.

5 Final discussion:

Even though LU-decomposition is slower than Thomas algorithm, it is very effiecient calculating a set of r linear equation with different right hand side. We can pre calculate the factorization of the initial matrix and keep the right hand side the same. This means we have to run the LU decomposition one time instead of r times compared to Thomas algorithm, but for this project clearly the specified and the general Thomas algorithm the most precise algorithm. See ([Hjorth-Jensen, 2015] section 6.4.1)

6 Conclusion:

As a concluding remark, it is crucial to examine the problem carefully. As we shown, due to the nature of the tridiagonal matrix it is more efficient and accurate to solve this with Gaussian elimination rather than using LU-decomposisition. For higher order of matrices LU-decomposition is very ineffictive and resulting memory lack. We also found out that increasing the number of mesh-point (over $N = 10^5$) we get round off errors and this goes as $\mathcal{O}(h^2)$. Thus it is ideal to choose good stepsize in order to produce good result with a relative fast speed.

References

[Hjorth-Jensen, 2015] Hjorth-Jensen, M. (2015). Lecture note fall.

[Mørken, 2015] Mørken, K. (2015). Nummerical algorithms and digital representation.

7 Appendix A:

In section 2.1 we introduced an expression for the nummerical second derivative of a given fucntion. In order to derive this equation, consider Taylor's theorem:

$$u(x) = \sum_{i=0}^{\infty} \frac{u^n(a)}{n!}(x-a) \tag{44}$$

We now want to expand the function $u(x)$ around the point $a+h$ and $a-h$ up to fourth order. We have now assumed that $u(x)$ is a smooth and continous function. Thus:

$$u(x \pm h) \approx u(x) \pm h\frac{\mathrm{d}u}{\mathrm{d}x} + \frac{h^2}{2!}\frac{\mathrm{d}^2u}{\mathrm{d}x^2} \pm \frac{h^3}{3!}\frac{\mathrm{d}^3u}{\mathrm{d}x^3} + \mathcal{O}(h^4)$$

We now want to add $u(x+h)$ and $u(x-h)$ together:

$$u(x+h) + u(x-h) = 2u(x) + h^2\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + \mathcal{O}(h^4) \tag{45}$$

Solving this with respect to the second derivative we get:

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2) \tag{46}$$

As we see this can also be written as:

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} \tag{47}$$

But we have to remember that the nummerical error goes as $\mathcal{O}(h^2)$.

8 Appendix B:

In section (2.3) we derived an expression for the diagonal elements for our tridiagonal matrix A. Recall our algorithm for triadiagonal matrix elements d_i:

$$d_i = b_i - c_{i-1}\frac{a_{i-1}}{d_{i-1}} \tag{48}$$

We will now impose that $a_i = c_i = -1$ and $b_i = 2$. Equation (48) will then look like:

$$d_i = 2 - \frac{1}{d_{i-1}} \tag{49}$$

Also keep in mind that we defined the initial condition as $d_1 = b_1 = 2$, thus:

$$d_2 = 2 - \frac{1}{2} = \frac{3}{2} \tag{50}$$

If we keep doing this up to $n = 5$ we have:

$$d_3 = \frac{4}{3} \tag{51}$$

$$d_4 = \frac{5}{4} \tag{52}$$

$$d_5 = \frac{6}{5} \tag{53}$$

Thus the diagonal element follows a pattern which goes as:

$$d_i = \frac{i+1}{i} \tag{54}$$