Numerical analysis of Two-point boundary value problems using Python

Vaishnavi Sharma, NIT Hamirpur

Report submitted for Summer Student Research Programme 2022 at TIFR-CAM

ABSTRACT

Differential equations are used to mathematically formulate the solutions of physical and other problems involving various variables. For example, Heat or sound propagation, fluid flows, electrostatics, elasticity, mathematical modelling for pandemic etc. Two-point Boundary value problems involve differential equations with boundary conditions at two points. Often, it is difficult to solve differential equations using analytical methods and hence we use numerical approach to solve them. This report includes numerical approach (Finite difference method) for solving boundary problems described under different schemes. It also describes different algorithms developed for finding the solution of linear systems formed after converting system of differential equations to algebraic equations.

1 INTRODUCTION

Numerical solutions for differential equations require conversion of differential equations to system of algebraic equations. The simplest way of generating such equations is to replace the derivatives in the equation by finite differences.The basic idea of any finite difference scheme comes from the definition of derivative of smooth function:

$$u'(x) = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h}$$

Therefore, in order to get a good approximation, h must be sufficiently small. Typically, the number of unknowns in the algebraic system is of order $O(1/h)$. We can consider differential equation as a linear system of infinitely many unknowns whose solution is known at endpoints and determined by differential equation in interior solution domain.

In order to define finite difference approximation for numerically solving partial differential equations, we will refer to Taylor's series that is used to provide approximations of derivatives. Assuming $f = f(x)$ is a four-times continuously differentiable function. For any $h > 0$, we have:-

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^3(x) + \frac{h^4}{24}f^4(x+h_1)$$

where h_1 is some number between 0 and h. Similarly,

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^3(x) + \frac{h^4}{24}f^4(x-h_2)$$

where h_2 is some number between 0 and h.

This implies that

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + E_h(x),$$

where error term E_h satisfies

$$|E_h(x)| \leq \frac{M_f h^2}{12}$$

where the constant M_f is given by:-

$$M_f = \sup_x |f^4(x)|$$

1.1 A system of algebraic equations

In order to understand the derivation of system of algebraic equations, we will take the below differential equation as an example.

$$-u''(x) = f(x), x \in (0,1), u(0) = u(1) = 0,$$

The first step is to divide the interval [0,1] into a finite number of subintervals. We introduce the grid points x_j, $j = 0, 1, 2...n$, given by:-

$$x_j = jh$$

where $n1$ is integer and spacing h is given by $h = 1/(n+1)$. The solution v is defined only at grid points x_j and approximated values are denoted by v_j.

Using the second order derivative defined in previous section, we can define the approximation v_j for $j = 0, 1, 2, ...n$ by:

$$-\frac{v_{j-1} - 2v_j + v_{j+1}}{h^2} = f(x_j)$$

for $j = 1, 2, ...n$, and $v_0 = v_{n+1} = 0$. This system of equations can be written in a more compact form by introducing the $n \times n$ matrix as follows:-

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

Further, let

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

where b_j is given by:-

$$b_j = h^2 f(x_j)$$

and

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Then, system of algebraic equations defined above can be written in compact form as:

$$Av = b$$

2 IMPLEMENTATION OF ABOVE DEFINED METHOD FOR A TWO-POINT BOUNDARY VALUE PROBLEM

Now, consider the boundary value problem

$$-u''(x) = f(x), u(0) = 0, u'(1) = 1.$$

where

$$f(x) = -e^{x-1}$$

We define two different schemes, S_1 and S_2 that approximate the solution of this problem. The differential equation and left boundary condition can be handled as usual, but the two schemes differ at approximation of second boundary condition. In the scheme S_1, we use the approximation

$$\frac{u_{n+1} - u_n}{h} = 1$$

and in S_2, we introduce an auxiliary unknown u_{n+2} and approximate the boundary condition by:

$$\frac{u_{n+2} - u_n}{2h} = 1$$

For both schemes, we define linear systems $A_1 v_1 = b_1$ and $A_2 v_2 = b_2$ respectively.

2.1 Scheme S_1

Approximate u'' using second order central difference

$$u_0 = a$$

$$-\frac{u_{j-1} - 2u_j + u_{j+1}}{h^2} = f_j, \qquad 1 \le j \le n$$

$$\frac{u_{n+1} - u_n}{h} = b$$

The boundary condition at $x = 1$ is only first order accurate because

$$\frac{u(x_{n+1} - u(x_n)}{h} = u'(x_{n+1}) + O(h)$$

Eliminating u_0, we get

$$-\frac{-2u_1 + u_2}{h^2} = f_1 + \frac{a}{h^2}$$

$$-\frac{u_{j-1} - 2u_j + u_{j+1}}{h^2} = f_j, \qquad 1 \le j \le n$$

$$-\frac{u_{n+1} - u_n}{h^2} = \frac{b}{h}$$

In matrix form, we can write it as:

$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 1 \end{bmatrix}\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_n \end{bmatrix} = \begin{bmatrix} f_1 + \frac{a}{h^2} \\ f_2 \\ \vdots \\ f_n \\ \frac{b}{h} \end{bmatrix}$$

2.2 Scheme S_2

We introduce an artificial/ghost point x_{n+2} and use central difference

$$U_0 = a$$

$$-\frac{U_{j-1} - 2U_j + U_{j+1}}{h^2} = f_j, \qquad 1 \le j \le n$$

$$\frac{U_{n+2} - U_n}{2h} = b$$

We want to eliminate U_{n+2}

$$-\frac{-2U_1 + U_2}{h^2} = f_1 + \frac{a}{h^2}$$

$$-\frac{U_{j-1} - 2U_j + U_{j+1}}{h^2} = f_j, \qquad 2 \le j \le n$$

$$-\frac{U_n - 2U_{n+1} + U_{n+2}}{h^2} = f_{n+1}$$

$$\frac{U_{n+2} - U_n}{2h} = b$$

which leads to

$$-\frac{-2U_1 + U_2}{h^2} = f_1 + \frac{a}{h^2}$$

$$-\frac{U_{j-1} 2U_j + U_{j+1}}{h^2} = f_j, \qquad 2 \le j \le n$$

$$\frac{U_{n+1} - U_n}{h^2} = \frac{1}{2} f_{n+1} + \frac{b}{h}$$

In matrix form, we have

$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 1 \end{bmatrix}\begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_n \\ U_{n+1} \end{bmatrix} = \begin{bmatrix} f_1 + \frac{a}{h^2} \\ f_2 \\ \vdots \\ f_n \\ \frac{1}{2} f_{n+1} + \frac{b}{h} \end{bmatrix}$$

We have the same matrix on the left, but the right hand side vector has an extra term in the last entry.

During the internship, I implemented a function $solve()$ that takes n and $scheme$ as inputs and output will be x(vector of grid points) and v(vector for approximated values of u).

Link for the code: Link of Python code developed for the above problem.

graphicx

After implementing the above code, we need to find its effectiveness by measuring the error produced between actual solution and numerical solution.

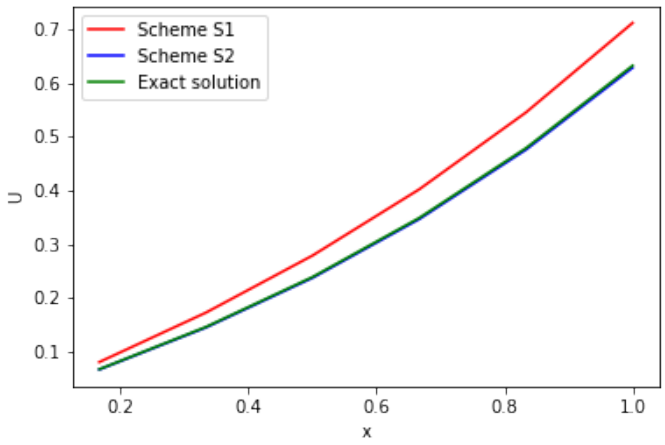

Figure 1. Numerical vs exact solution for both schemes S1 and S2. We can observe that Scheme S2 is a better approximation.

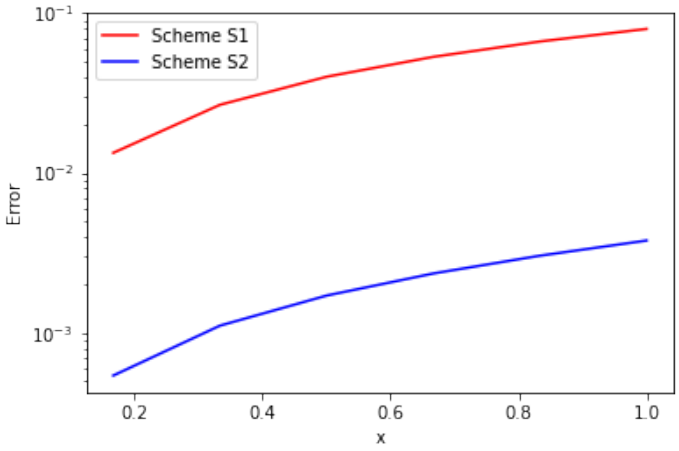

Figure 2. Semilog plot for Error vs x. Scheme S1 has larger values of error with respect to x in comparison to scheme S2.

2.3 Error Calculation

Error for a specific value of h can be calculated by:

$$E_h = max|u(x_j) - U_j|, \qquad 0 \leq j \leq n+1$$

graphicx After plotting E_h vs h, we will observe that, E_h satisfies a bound of the form:

$$E_h = O(h^2)$$

2.4 Convergence matrix

Finally, in order to find the rate of convergence of computed values towards original function for both schemes, Rate of convergence α can be calculated by:

$$\alpha = \frac{log(e_{h_1}/e_{h_2})}{log(h_1/h_2)}$$

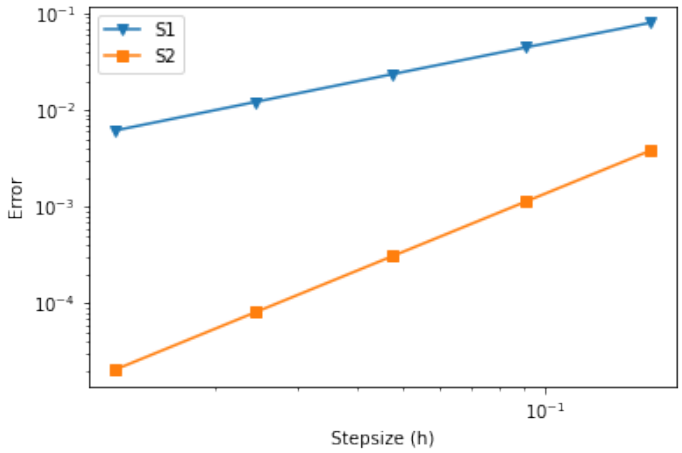

Figure 3. loglog plot for Error vs h. Slopes of both lines denote the rate of convergence towards original solution. Hence, Scheme S2 has higher rate of convergence with respect to h.

where

$$h_1 = \frac{1}{n_1 + 1}$$

and

$$h_2 = \frac{1}{n_2 + 1}$$

graphicx

3 ALGORITHM TO DEVELOP THE SOLUTION OF LINEAR SYSTEM OF THE FORM $AV = B$

In the above boundary value problems, we have used *scipy.linalg* module to find the solution vector y. In this section, we will develop an algorithm for solving such linear systems of the form $Av = b$.

3.1 Problem statement

Given a linear system:

$$Av = b$$

where A is a tri-diagonal $(n \times n)$ matrix of the form:

$$A = \begin{bmatrix} \alpha_1 & \gamma_1 & 0 & \dots & 0 \\ \beta_2 & \alpha_2 & \gamma_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \beta_{n-1} & \alpha_{n-1} & \gamma_{n-1} \\ 0 & \dots & 0 & \beta_n & \alpha_n \end{bmatrix}$$

v is a $(n \times 1)$ vector:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

b is $(n \times n)$ vector:

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

During the internship, I had written a function $solve()$ that will take α, β, γ and b as inputs and will give v as output. This algorithm is based on *Gaussian Elimination* for tri-diagonal linear systems.

3.2 Brief overview of Gaussian elimination

Linear system described above can be equivalently written as:

$$\alpha_1 v_1 + \gamma_1 v_2 = b_1,$$

$$\beta_2 v_1 + \alpha_2 v_2 + \gamma_2 v_3 = b_2,$$

$$\beta_3 v_2 + \alpha_3 v_3 + \gamma_3 v_4 = b_3,$$

$$\vdots$$

$$\beta_{n-1} v_{n-2} + \alpha_{n-1} v_{n-1} + \gamma_{n-1} v_n = b_{n-1},$$

$$\beta_n v_{n-1} + \alpha_n v_n = b_n$$

We use first equation to eliminate the first variable v_1 from second equation. Then, new version of second equation will be used to eliminate the second variable (v_2) from third equation and so on. If we subtract $m_2 = \beta_2/\alpha_1$ times the first equation from second equation, second equation becomes:

$$\delta_2 v_2 + \gamma_2 v_3 = c_2$$

where

$$\delta_2 = \alpha_2 - m_2 \gamma_1$$

$$c_2 = b_2 - m_2 b_1$$

In general, if $(j-1)th$ equation has been transformed as:

$$\delta j - 1 v_{j-1} + \gamma_{j-1} v_j = c_{j-1}$$

and the original j_{th} equation is:

$$\beta_j v_{j-1} + \alpha_j v_j + \gamma_j v_{j+1} = b_j$$

If $m_j = \beta_j/\delta_{j-1}$ times transformed $(j-1)_{th}$ equation is subtracted from original j_{th} equation, we get:

$$\delta_j v_j + \gamma_j v_{j+1} = c_j$$

$$\delta_j = \alpha_j - m_j \gamma_{j-1},$$

$$c_j = b_j - m_j c_{j-1}$$

Hence, the variables that need to be calculated in the algorithm can be defined as:

$$\delta_1 = \alpha_1, c_1 = b_1$$

$$m_j = \beta_j/\delta_{j-1}$$

$$\delta_j = \alpha_j - m_j \gamma_{j-1}, \qquad 2 \le j \le k$$

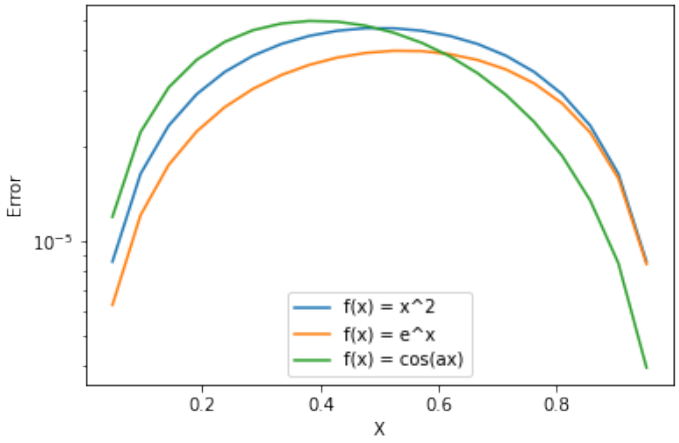

Figure 4. Semi-log Plot for Error vs x for different functions f(x). Error increases upto a certain value of x and starts decreasing after that point and hence follows a parabolic routine.

$$c_j = b_j - m_j c_{j-1}$$

Link for the code: Link of Python code written for the above problem.

3.3 Implementation of above code for some right hand side functions $f(x)$

We now narrow down the matrix A to the coefficients matrix for second order finite difference scheme for the differential equation defined below:

$$-u''(x) = f(x), x \in (0,1)$$

Subject to the conditions:

$$u(0) = u(1) = 0$$

As described in section 2, we form a linear system of the form:

$$\frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} f_[1] \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

Some functions $f(x)$ for which we solve this linear system are:

$$f(x) = x^2$$

$$f(x) = e^x$$

$$f(x) = cos(ax)$$

graphicx For error calculation, refer to section 2.3. Code for the above implementation: Link of Python code written for the above problem.

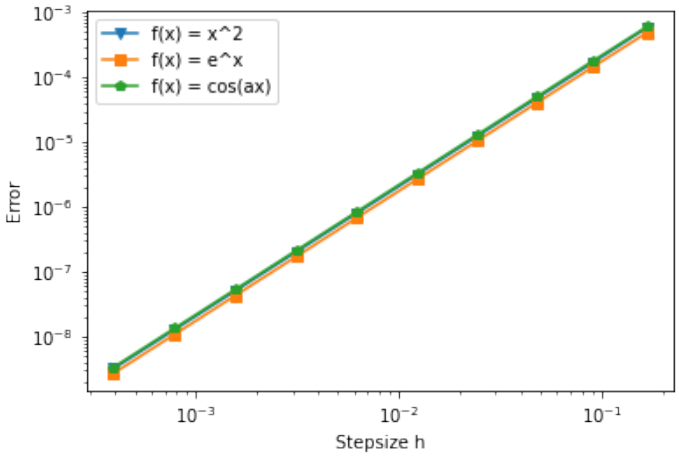

Figure 5. loglog Plot for Error vs x for different functions f(x). Error increases with h linearly for all three right hand side functions f(x) defined.

3.4 Modified algorithm for series of linear systems

Often, we want to solve many linear systems of the form $Av = b$ at once in which matrix A remains fixed but right hand side vector b changes (and hence vector v changes). If we want to solve

$$Av_l = b_l$$

for $l = 1, 2, ...N$ To solve such systems, we modify the function according to a theorem called $LU\ Decomposition$.

3.5 Brief overview of LU decomposition

LU Decomposition can be said as another form of gaussian elimination as we eliminate the variables to form the upper triangular and lower triangular matrices in gaussian elimination. According to LU Decomposition, Given a matrix $A \in R^{m\times n}$ with mn, its $LU\ Decomposition$ is given by $A = LU$ where $L \in R^{m\times n}$ is a unit lower triangular matrix and $U \in R^{n\times n}$ is upper triangular with non-zeros on its diagonal.

We define a function called $LU_decomposition()$ that will output the matrix L and U for matrix A. Then we call the function $solve()$ that takes diagonals of L and U and vector b as inputs and gives solution vector v as output.

For tri-diagonal matrices, L and U are defined as:

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_1 & 1 & 0 & \dots & 0 \\ 0 & l_2 & 1 & \dots & 0 \\ \vdots & 0 & \ddots & \ddots & 0 \\ 0 & 0 & \dots & l_n & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} u_1 & \gamma_1 & 0 & \dots & 0 \\ 0 & u_2 & \gamma_2 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & u_{n-1} & \gamma_n \\ 0 & \dots & \dots & 0 & u_n \end{bmatrix}$$

3.6 Brief overview of algorithm

Algorithm for $LU\ Decomposition$ can be described as follows:

$$u_1 = \alpha_1$$

$$\text{Solve for } l_i : l_{i-1}u_{i-1} = \beta_{i-1},$$

$$u_i = \alpha_i - (l_{i-1} * (\gamma_{i-1})), \qquad 2 \le i \le n$$

3.7 $Solve()$ function algorithm

After obtaining L and U, $Av = b$ can be rewritten as:

$$LUv = b$$

This can further be rewritten as:

$$Lx = b$$

where

$$x = Uv$$

Above equation can be solved for x by forward substitution as follows:

$$x_1 = b_1$$

$$x_i = b_i - (l_{i-1} * x_{i-1}), \qquad 2 \le i \le n$$

After obtaining x, we will solve

$$Uv = x$$

by backward substitution as follows:

$$v_n = x_n/u_n$$

for i = N-1:-1:-1 (reverse loop)

Solve for v_i:

$$u_i v_i : x_i - (\gamma_i * x_{i+1})$$

3.8 Implementing above modified algorithm for $f(x) = e^{x/l}$

We now implement the above modified algorithm for solving the following problems:

$$-u''(x) = e^{x/l}, x \in (0, 1)$$

subject to the conditions:

$$u(0) = u(1) = 0$$

for $l = 1, 2, ...10$. by equating right hand side vector b to $e^{x/l}$.

graphicx

4 RESULTS AND CONCLUSION

We now list down the observations of above implemented algorithms and what we can conclude from these observations.

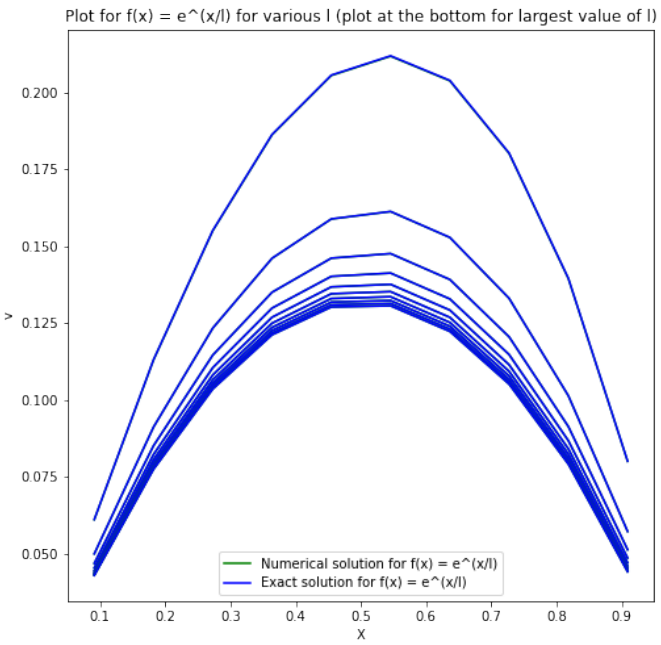

Figure 6. Numerical vs exact solution plot for different f(x) = $e^{x/l}$. As l increases, the curves become similar and it becomes difficult to differentiate between them.

4.1 Analysis of both schemes in section 2 problem

From *Figure* 3, we can observe that the rate of convergence of Scheme $S2$ towards exact solution is more than that of Scheme $S1$. This implies that Scheme $S2$. On further investigation, we find that Scheme $S2$ is a second order central difference approximation for right hand side boundary point:

$$\frac{U_{n+2} - U_n}{2h} = b$$

While for Scheme $S1$, we used second order backward difference approximation for right hand side boundary point:

$$\frac{U_{n+1} - U_n}{h} = b$$

If we refer to $Taylor's\ series$ described in Introduction part, we will find that we can find the accuracy of certain scheme using order of approximation that we have used. In scheme S1, the truncation error of the backward difference approximation has the order of $O(h)$.

While in scheme S2, the truncation error of central difference approximation has the order of $O(h^2)$. Clearly, this implies that Scheme $S2$ gives better approximation to exact solution and hence has higher rate of convergence (denoted by slope of loglog plot of error vs h).

4.2 Analysis of algorithm implemented in section 3

In section 3, we developed an algorithm for solution of linear system of the form $Av = b$ based on Gaussian elimination. Then that algorithm was implemented to solve two-point boundary value problems for some right hand side functions given below:

$$f(x) = x^2$$

$$f(x) = e^x$$

$$f(x) = cos(ax)$$

Often, we want to solve a series of linear systems at once in which right hand side vector b changes while matrix A remains same (As described in section 3.5). For that we modified the algorithm a bit by introducing $LU\ Decomposition$ theorem. LU Decomposition returned the diagonals of matrices L and U which were given to $solve()$ function as input to perform backward and forward substitution in order to output the solution vector v.

We used this modified algorithm to solve series of linear systems given by $f(x) = e^{x/l}$ for l = 1,2,...N. After plotting the numerical vs exact solutions for all values of l, we observed that as the value of l is increasing, the curves are becoming similar and the variation is tending towards 0. From this, we can then approximate the numerical solution for $f(x) = e^{x/}$ as $l \to \infty$.

4.3 Conclusion

From the above analysis, we can conclude that finite difference approximation to differential equations give us good results and that the equations become easier to solve.

5 ACKNOWLEDGEMENT

I would like to take this opportunity to thank my guide Prof. Praveen Chandrashekar who introduced me to this interesting field of numerical analysis of Differential equations. This internship helped me to take my first formal step in the field of research and also helped me to develop an insight on how research is carried out. I also got to know the importance of focusing on fine details in order to work out any problem effectively and that how research requires patience and discipline in order to get good results.

6 REFERENCES

Some references for the above work:

1. Introduction to partial differential equations: A computational approach, Aslak Tveito and Ragnar Winther: https://link.springer.com/book/10.1007/b138016

2. Matrix Computations, Gene H Golub and Charles F. Van Loan: https://www.press.jhu.edu/books/title/10678/matrix-computations

3. A short introduction to Python for computing: https://github.com/cpraveen/python

4. Latex-A document preparation system: https://www.latex-project.org/