HW 0 CFD

Roi Baruch, ID: 211914866

June 15, 2024

Contents

List of Figures

1 Introduction

The following report is on HW0 in the Computational Fluid Dynamics course 086376. The homework consists of using a finite difference method to solve a second-order ODE numerically. The report shows the different boundary conditions, mesh sizes, and precision types. The numerical solver is implemented in C and compiled, run, and plotted using Matlab.

2 The mathematical problem

Consider the following second-order differential equation:

$$y'' + xy' + x^2 y = \sin(2\pi x) + \cos(2\pi x) \tag{1}$$

This is a linear non-homogeneous 2^{nd} order ODE. To get a solution, we need the boundary conditions:

1. Dirichlet boundary conditions:

$$y(0) = 0 \tag{2}$$

$$y(1) = 1 \tag{3}$$

2. Neumann boundary conditions:

$$y'(0) = 1 \tag{4}$$

$$y'(1) = -1 \tag{5}$$

3 Numerical scheme

3.1 Finite Difference Formulation

Consider the general second-order differential equation:

$$a(x)y'' + b(x)y' + c(x)y = d(x) \tag{6}$$

in the interval $[a, b]$. Let h be the cell size given by:

$$h = \frac{b-a}{N} \tag{7}$$

Where N is the number of equispaced cells. Therefore x_i is given by:

$$x_i = a + i \times h \tag{8}$$

Using finite difference operators, the equation may be discretized as follows:

$$a(x_i)\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + b(x_i)\frac{y_{i+1} - y_{i-1}}{2h} + c(x_i)y_i = d(x_i) \tag{9}$$

Rearranging the above discrete form results in the following:

$$\left[\frac{a(x_i)}{h^2} - \frac{b(x_i)}{2h}\right] y_{i-1} + \left[-\frac{2a(x_i)}{h^2} + c(x_i)\right] y_i + \left[\frac{a(x_i)}{h^2} + \frac{b(x_i)}{2h}\right] y_{i+1} = d(x_i) \tag{10}$$

Or in a concise form:

$$A(x_i)y_{i-1} + B(x_i)y_i + C(x_i)y_{i+1} = D(x_i) \tag{11}$$

where

$$A(x_i) = \frac{a(x_i)}{h^2} - \frac{b(x_i)}{2h} \tag{12}$$

$$B(x_i) = -\frac{2a(x_i)}{h^2} + c(x_i) \tag{13}$$

$$C(x_i) = \frac{a(x_i)}{h^2} + \frac{b(x_i)}{2h} \tag{14}$$

so that

$$A_i = A(x_i) \tag{15}$$

$$B_i = B(x_i) \tag{16}$$

$$C_i = C(x_i) \tag{17}$$

3.2 Dirichlet Boundary Conditions

The boundary conditions are given as:

$$y(a) = Y_0 \tag{18}$$

$$y(b) = Y_N \tag{19}$$

The equation at $i = 1$ takes the form:

$$B_1 y_1 + C_1 y_2 = D_1 - A_1 Y_0 \tag{20}$$

The equation at $i = N - 1$ takes the form:

$$A_{N-1} y_{N-2} + B_{N-1} y_{N-1} = D_{N-1} - C_{N-1} Y_N \tag{21}$$

Linear System

The linear system that results from the discretization and the implementation of the Dirichlet boundary conditions is as follows:

$$\begin{pmatrix} B_1 & C_1 & 0 & \cdots & \cdots & 0 & 0 \\ A_2 & B_2 & C_2 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 & \cdots & 0 \\ 0 & 0 & A_i & B_i & C_i & 0 & \\ 0 & \cdots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & A_{N-2} & B_{N-2} & C_{N-2} \\ 0 & 0 & \cdots & \cdots & 0 & A_{N-1} & B_{N-1} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ \cdots \\ \cdots \\ y_{N-2} \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} D_1 - A_1 Y_0 \\ D_2 \\ \cdots \\ \cdots \\ \cdots \\ D_{N-2} \\ D_{N-1} - C_{N-1} Y_N \end{pmatrix}$$

The solution is obtained through a direct inversion using the Thomas algorithm.

3.3 Neumann Boundary Conditions

The boundary conditions are given as:

$$y'(a) = Y'_0 \tag{22}$$

$$y'(b) = Y'_N \tag{23}$$

In this case, the equations at both ends of the interval are added to the linear system using the ghost cell concept. The equation at $i = 0$ takes the form:

$$A_0 y_{-1} + B_0 y_0 + C_0 y_1 = D_0 \tag{24}$$

The equation at $i = N$ takes the form:

$$A_N y_{N-1} + B_N y_N + C_N y_{N+1} = D_N \tag{25}$$

Applying the boundary conditions results in:

$$y'(0) = \frac{y_1 - y_{-1}}{2h} = Y'_0 \tag{26}$$

$$y'(N) = \frac{y_{N+1} - y_{N-1}}{2h} = Y'_N \tag{27}$$

The ghost cells are then evaluated using the following:

$$y_{-1} = y_1 - 2hY'_0 \tag{28}$$

$$y_{N+1} = y_{N-1} + 2hY'_N \tag{29}$$

Subsequently, the ghost cells are substituted into the system.

Linear System

The linear system that results from the discretization and the implementation of the Neumann boundary conditions is as follows:

$$\begin{pmatrix} B_0 & A_0+C_0 & 0 & \cdots & \cdots & 0 & 0 \\ A_1 & B_1 & C_1 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 & \cdots & 0 \\ 0 & 0 & A_i & B_i & C_i & 0 & 0 \\ 0 & \cdots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & A_{N-1} & B_{N-1} & C_{N-1} \\ 0 & 0 & \cdots & \cdots & 0 & A_N+C_N & B_N \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \cdots \\ \cdots \\ \cdots \\ y_{N-1} \\ y_N \end{pmatrix} = \begin{pmatrix} D_0+2hA_0Y_0' \\ D_1 \\ \cdots \\ \cdots \\ \cdots \\ D_{N-1} \\ D_N-2hC_NY_N' \end{pmatrix} \tag{30}$$

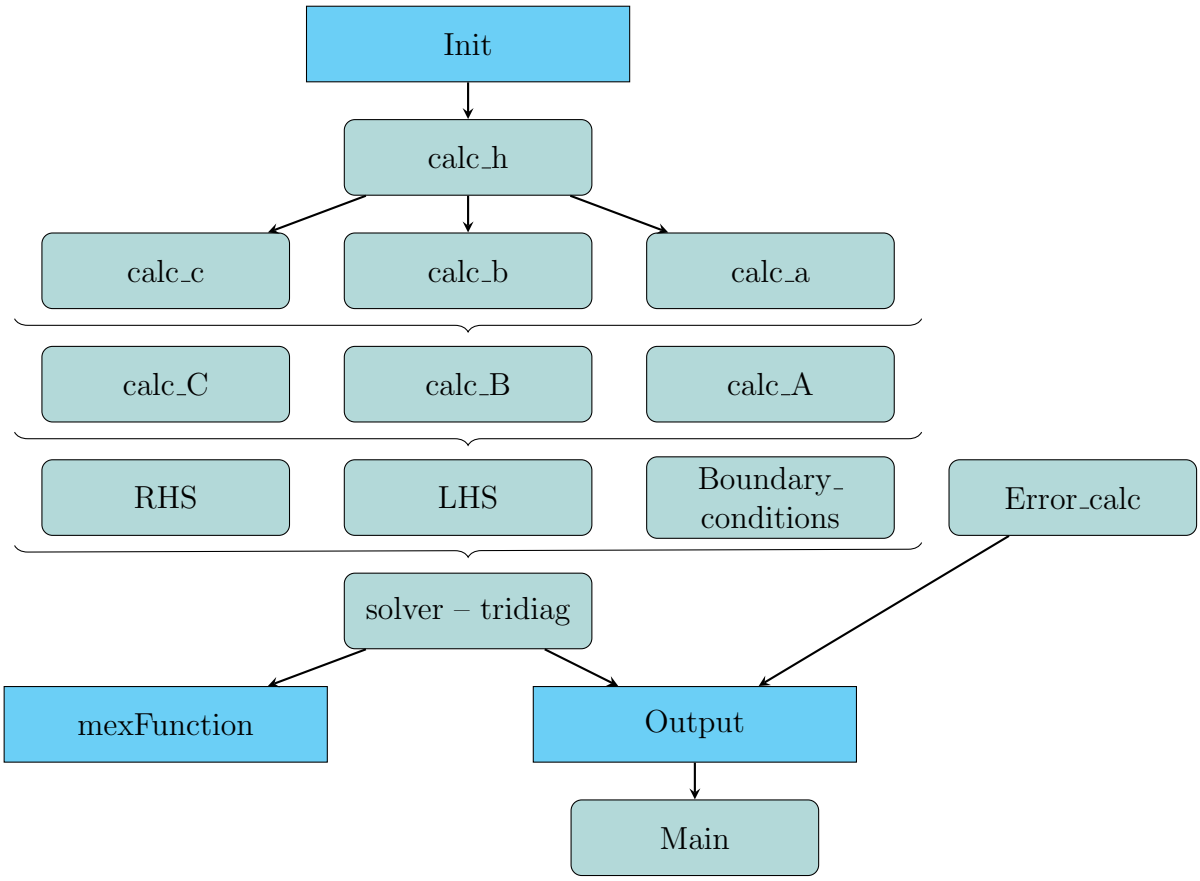
The solution is obtained through a direct inversion using the Thomas algorithm.

4 Functional description

My program consists 2 parts:

1. The C program that numerically solves the ODE
2. A Matlab script that runs the C program as detailed in 4.2.2.

4.1 C Program flowchart



4.2 List of functions

4.2.1 C

The following is a list of the functions implemented in C:

1. Init - The function reads the input file in the given directory and initializes the problem's parameters.
2. calc_h - Calculates the step size h for a given mesh size - N, and the wanted interval [a b].
3. calc_a - Initialize array 'a' to 1 - $a(x_i)y'' = y''$
4. calc_b - Initialize array 'b' to x - $b(x_i)y' = xy'$
5. calc_c - Initialize array 'c' to x^2 - $c(x_i)y = x^2y$
6. calc_A - Initialize the lower diagonal array 'A' to $A(x_i) = \frac{a(x_i)}{h^2} - \frac{b(x_i)}{2h}$
7. calc_B - Initialize the middle diagonal array 'B' to $B(x_i) = -\frac{2a(x_i)}{h^2} + c(x_i)$
8. calc_C - initialize the upper diagonal array 'C' to $C(x_i) = \frac{a(x_i)}{h^2} + \frac{b(x_i)}{2h}$
9. LHS - calls the diagonals for the LHS of the function.
10. RHS - initialize array 'd' for the RHS of the function to $d(x) = \sin(2\pi x) + \cos(2\pi x)$
11. Boundary conditions - applies the boundary_conditions to all the relevant arrays for the given boundary conditions.
12. error_calc - initialize A, B, and C vectors for error calculation.
13. tridiag - The given solver.
14. output - writes the output files, used when the code runs through the compiler, not Matlab.
15. Main/mexFunction - the main function when all the functions are called, and all the relevant operations are performed. Main is used when the code is compiled and run through VS code (or Clion or any other compiler), and mexFunction is used when the code is compiled and run through Matlab.

4.2.2 Matlab

The following is a list of sections in the Matlab script:

1. Input parameters - here, the user manually enters the input variables (boundary condition, the interval, N values, and the precision).

2. Writing the input.txt file - this section takes the variables from the previous section and writes an input file for the C code to read in a .txt format.

3. Running the C program - This section runs the corresponding C code for either (either using float or double for all the variables in the C program) and inputs all the needed data to plot the graphs and the errors.

4. Ploting the function - This section plots the corresponding graph.

5. Error calculation - This section calculates the error for either Dirichlet or Neumann boundary conditions and plots them on a logarithmic scale.

6. boundary conditions - This section adds to the plots the boundary condition for the Neumann boundary condition case.

5 Results

5.1 Dirichlet boundary conditions

5.1.1 General solution

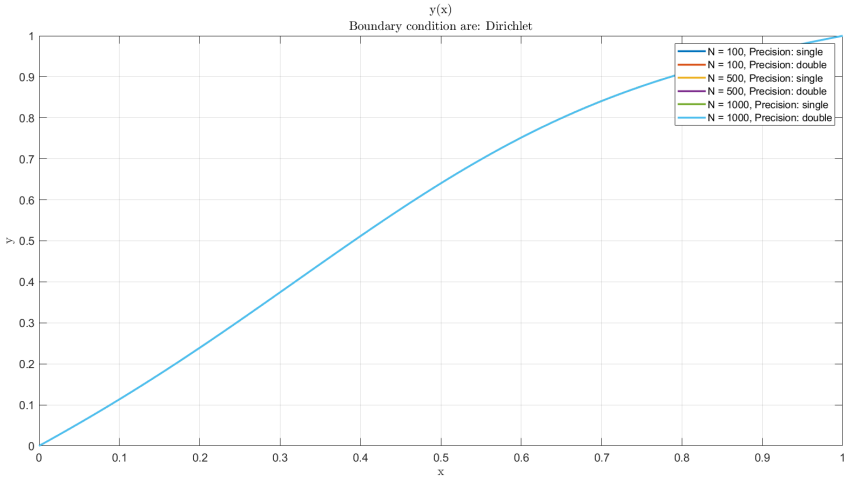


Figure 1: $y(x)$ for Dirichlet boundary condition

In figure 1, we can see the general solution of our function for a Dirichlet set of boundary conditions. At first glance, we can observe that the boundary conditions $y(0) = 0$ and $y(1) = 1$ were satisfied, and the differences between mesh size and single or double precision appear negligible.

5.1.2 Mesh size

From figure 1, we can see that for an initial test for mesh sizes of 100, 500, and 1000 cells for the interval of [0 1], no difference can be observed. Zooming in, we can notice minor differences:

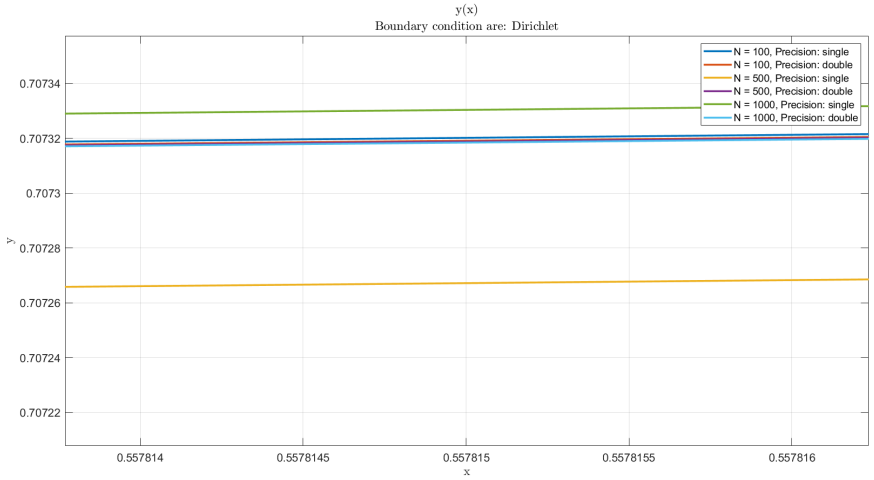


Figure 2: $y(x)$ for Dirichlet zoom in

Zooming in differences in the order of $10e^{-4}$ can be observed; the graphs with single precision and higher N values can be seen deviating from the rest. Further expanding the N values, a trend can be observed:

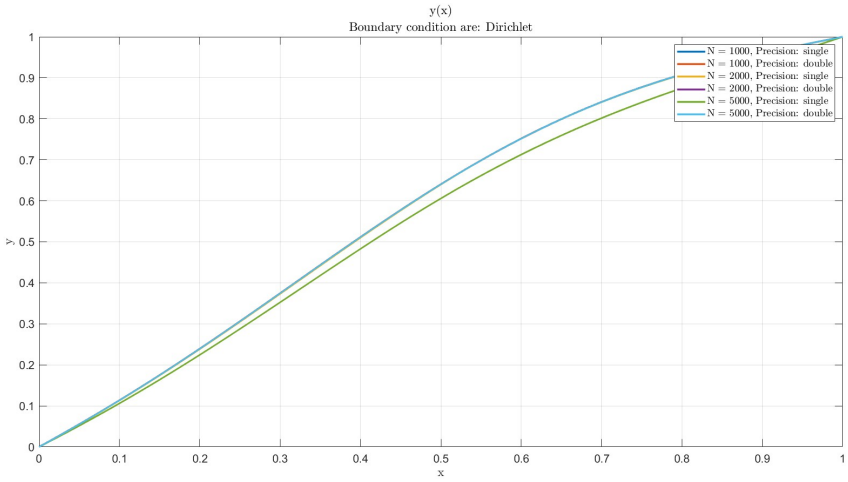


Figure 3: $y(x)$ for Dirichlet with high N values

As the mesh size decreases (N values increase), the accuracy of the solution decreases. For $N = 5000$, we can see a significant difference in single precision (the green line in figure 3).

5.1.3 Single vs. double precision

Now, comparing between single and double precision, using float a variable for single precision vs a double variable for double precision, a trend can be observed:

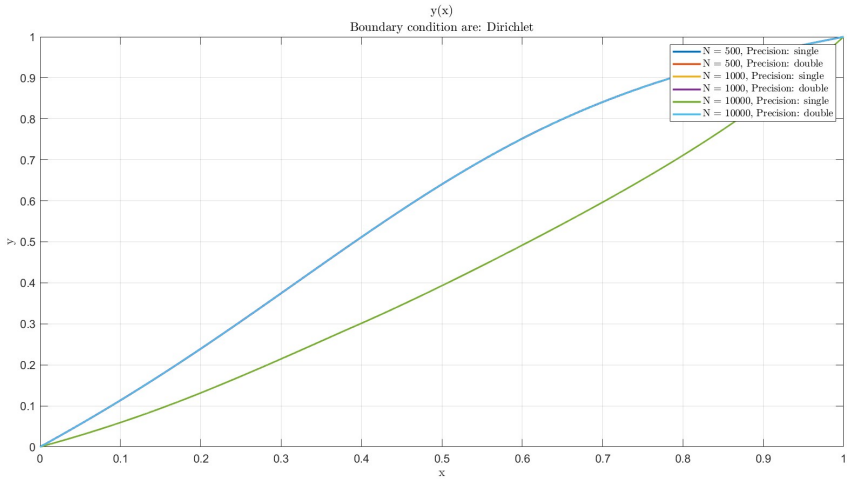


Figure 4: $y(x)$ for Dirichlet with N = 10k

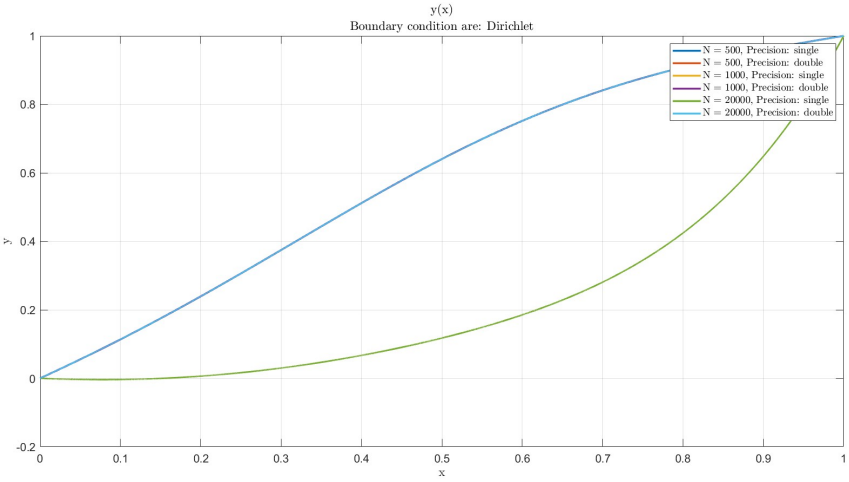


Figure 5: $y(x)$ for Dirichlet with N = 20k

In both figure 4 and figure 5, the same trend continues. As the N value increases, the single precision graph deviates from the nominal state, suggesting that a single precision approach isn't sufficient for the problem. To further investigate, we can plot the error of the graph. The error is calculated as follows:

$$LHS * y = RHS \tag{31}$$

$$LHS * y - RHS = Error \tag{32}$$

So, we'll define the error as the absolute:

$$Error = |LHS * y - RHS| \tag{33}$$

Now, plotting the error with regards to x on a logarithmic scale:

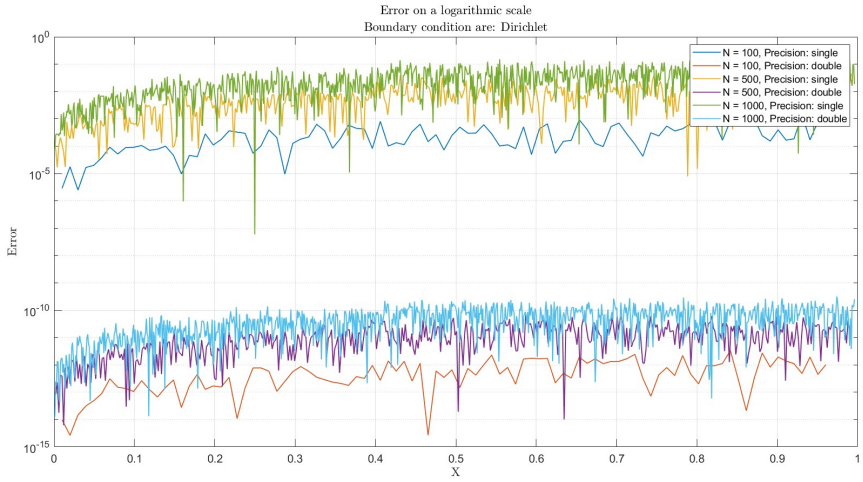


Figure 6: Error for Dirichlet

In figure 6, we see a clear difference between single and double precision. While a higher N value corresponds to a higher error for both, the average error for double precision is about **10** orders of magnitude lower than single precision.

5.2 Neumann boundary conditions

5.2.1 General solution

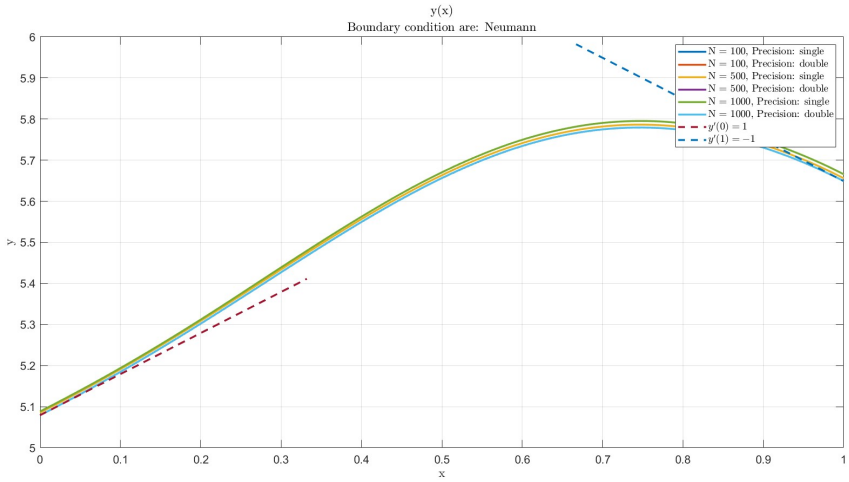


Figure 7: $y(x)$ for Neumann boundary condition

As expected, in figure 7, we can see a different solution to the function from the Dirichlet boundary conditions shown in figure 1. At first glance, we can see that the solution satisfies the boundary conditions given (the dotted lines as written in the legend).

5.2.2 Boundary conditions

Zooming in to check the boundary conditions, we see:

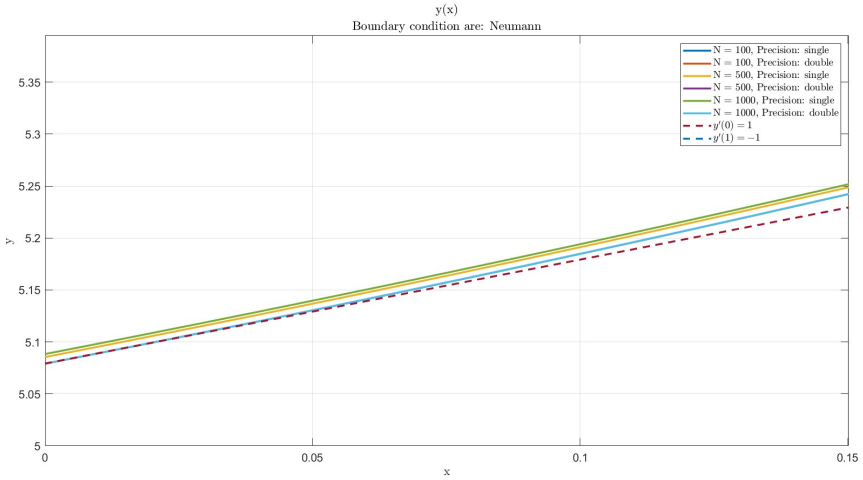


Figure 8: $y(x)$ at x = 0

In figure 8, we can see that all the plots satisfy the boundary condition, and for higher N values and double precision, the solution becomes more "precise," to be further checked in 5.2.4.

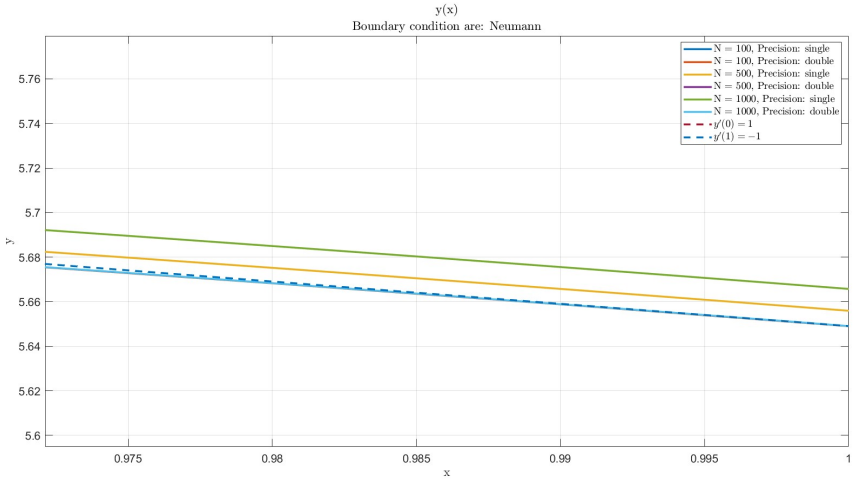


Figure 9: $y(x)$ at x = N

In figure 9, we can see a similar behavior to figure 8. The solution does match the boundary condition.

5.2.3 Mesh size

Further checking the mesh size (N values), we can zoom in to figure 1:

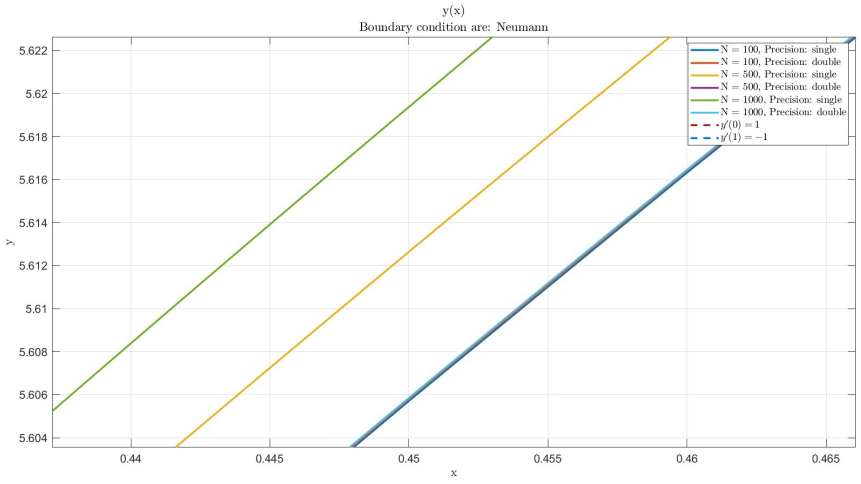


Figure 10: $y(x)$ for Neumann boundary condition zoom in

In figure 10, we can see that the solution converges to a single line for higher N values. Further increasing the N values:

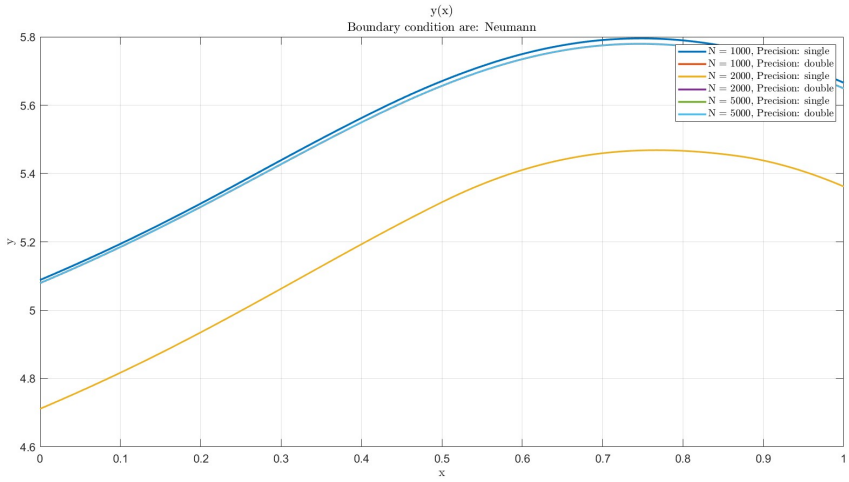


Figure 11: $y(x)$ for Neumann with high N values

We can see in figure 11 that for higher N values, the solution diverges or is even non-existent for N = 5000 for both single or double precision.

5.2.4 Single vs. double precision

Checking the difference between single and double precision, we'll plot the graphs for N = 1000, 2000:

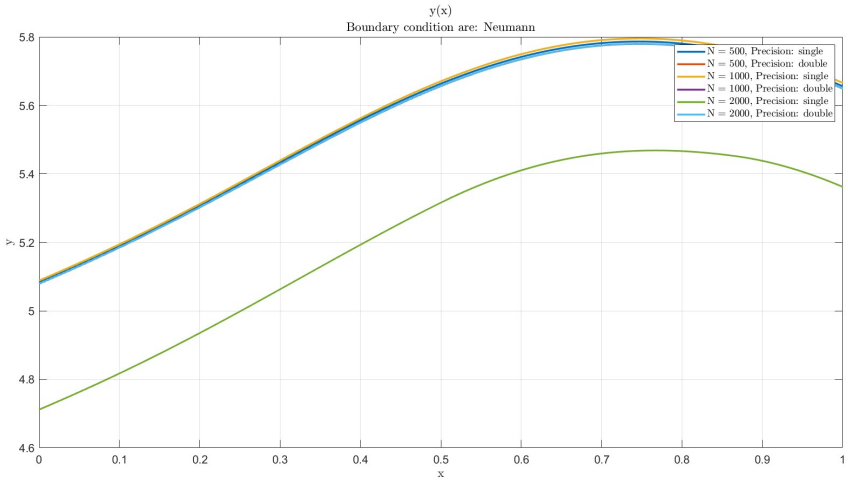


Figure 12: $y(x)$ for Neumann single vs. double precision

In figure 12, we can see the difference between single and double precision for a set of Neumann boundary conditions is much more drastic than for a set of Dirichlet boundary conditions, as shown in 5.1.3. For N = 2000, the solution drastically changes from the nominal, while the double precision remains at the nominal. Now, looking at the error as calculated in equation 33:

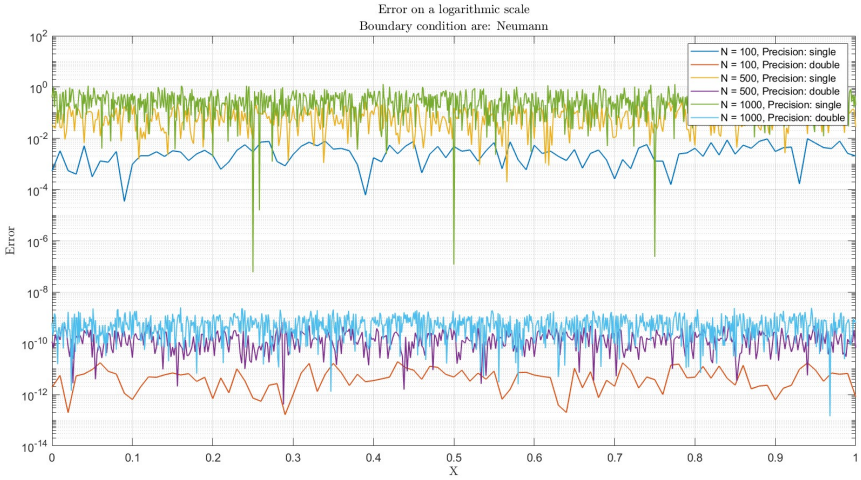


Figure 13: Error for Neumann

In figure 13, we can see that the error for double precision is lower by around **10** orders of magnitude than the single precision error.

6 Conclusions

In this homework exercise, we wrote a C program to solve a second-order ODE using a finite differences method. The effects of the boundary conditions, mesh size, and single vs. double precious. The main conclusions are:

- As shown in 5.1.2 and 5.2.3, the solution is much more sensitive to the mesh size for a set of Neumann boundary conditions than for Dirichlet.
- For each solver use and a given tolerance wanted in the solution, the N value can be optimized using the error plots.
- Using double precision seems more beneficial in the Neumann case than the Dirichlet case. This comes from the high sensitivity of the Neumann case to the mesh size than the Dirichlet one.

7 Appendix

7.1 C program

The following code is for a single precision case; in a double precision case, all the functions and variables were changed from float to double.

```
    /* 7/6/24, Roi Baruch
This code id HW0 in Computational Fluid Dynamics course No.
    086376
link to Git repository: https://github.com/roibaruch24/
    CFD_086376
*/
// Including all the relevent headers:
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "mex.h"
// Init - reads the input file and initializes all the relevent
    valuse
int Init(int *N, char *boundary_condition, float *start,float *
    end,float *bc_0,float *bc_n){
    const char *input_file_path = "C:\\Users\\roiba\\Documents\\
        CFD_086376\\HW0\\input.txt";
    FILE *input = fopen(input_file_path, "rt");
    if (input == NULL) {
        return 1;
    }
    fscanf(input,"%d-%f-%f-%c-%f-%f",N,start,end,
        boundary_condition,bc_0,bc_n);
    fclose(input);
    return 0;
}
// calc_h - calculates the step value h
void calc_h(float *h, float start, float end, int N)
{
    *h=(float)(end - start) / N;
}
// calc_a - initialize array 'a' to 1
void calc_a(float *a, int N) {
    for (int i = 0; i <= N; i++) {
        a[i] = 1;
    }
}
```

```
// calc_b - initialize array 'b' to x
void calc_b(float *b,float h,int N)
{
    for (int i = 0;i<=N; i++)
    {
        b[i]=i*h;
    }
}
// calc_c - initialize array 'c' to x^2
void calc_c(float *c,float h,int N)
{
    for (int i = 0;i<=N; i++)
    {
        c[i]=(i*h)*(i*h);
    }
}
// calc_A - initialize the lower diagonal array 'A'
void calc_A(float *A_dig, float *a,float *b,float h,int N)
{
    for (int i = 0; i<=N; i++)
    {
        A_dig[i]=a[i]/(h*h) - b[i]/(2*h);
    }
}
// calc_B - initialize the middle diagonal array 'B'
void calc_B(float *B_dig,float *a,float *c,float h,int N)
{
    for (int i = 0; i<=N; i++)
    {
        B_dig[i]=-2*a[i]/(h*h) +c[i];
    }
}
// calc_C - initialize the upper diagonal array 'C'
void calc_C(float *C_dig,float *a,float *b,float h,int N)
{
    for (int i = 0; i<=N; i++)
    {
        C_dig[i]=a[i]/(h*h) +b[i]/(2*h);
    }
}
// LHS - calls the the diagonals for the LHS of the function
void LHS (float *a, float *b, float *c, float *A_dig, float *
   B_dig, float *C_dig, float h, int N)
{
    calc_A(A_dig, a, b, h, N);
```

```
    calc_B(B_dig, a, c, h, N);
    calc_C(C_dig, a, b, h, N);
}
// RHS - initialize array 'd' for the RHS of the function
void RHS(float *d,float h,float N)
{
    for (int i = 0; i<=N; i++)
    {
        double x = 2*3.14159265358979323846264338327950288*i*h;
        d[i] = sin(x) + cos(x);
    }
}
// Boundary_conditions - applies the boundary conditions
void Boundary_conditions(float *A_dig, float *C_dig, float *d,
    int N, float h, int *is, int *ie, float *u, char
    boundary_condition, float bc_0, float bc_N) {
    if (boundary_condition == 'D')
    {
        *is = 1;
        *ie = N-1;
        d[*is] -= A_dig[*is]*bc_0;
        d[*ie] -= C_dig[*ie]*bc_N;
        u[0] = bc_0;
        u[N] = bc_N;

    }
    else if (boundary_condition == 'N')
    {
    *is = 0;
    *ie = N;
    C_dig[*is] += A_dig[*is];
    A_dig[*ie] += C_dig[*ie];
    d[*is] += 2*h*A_dig[*is]*bc_0;
    d[*ie] -= 2*h*C_dig[*ie]*bc_N ;
    }
}
// initialize A, B, C vectors for error calculation
void error_calc(float *A_dig, float *B_dig, float *C_dig, float
    *d, float *A_dig_er, float *B_dig_er, float *C_dig_er, float
    *d_er, int N ){
    for (int i = 0; i<=N; i++)
    {
    A_dig_er[i] = A_dig[i];
    B_dig_er[i] = B_dig[i];
    C_dig_er[i] = C_dig[i];
```

```
    d_er[i] = d[i];
    }
}
// Tridiag - is the solver
int tridiag(float *A_dig, float *B_dig, float *C_dig, float *d,
    float *u, int is, int ie)
{
  int i;
  float beta;
  for (i = is + 1; i <= ie; i++)
    {
      if(B_dig[i-1] == 0.) return(1);
      beta = A_dig[i] / B_dig[i-1];
      B_dig[i] = B_dig[i] - C_dig[i-1] * beta;
      d[i] = d[i] - d[i-1] * beta;
    }

  u[ie] = d[ie] / B_dig[ie];
  for (i = ie - 1; i >= is; i--)
    {
      u[i] = (d[i] - C_dig[i] * u[i+1]) / B_dig[i];
    }
  return(0);
}
// output - writes the output files
/*int output (int N, float h, float *u, float *A_dig, float *
    B_dig, float *C_dig)
{
    const char *solution_file_path = "C:\\Users\\roiba\\
        Documents\\CFD_086376\\HW0\\solution.dat";
    const char *error_calc_file_path = "C:\\Users\\roiba\\
        Documents\\CFD_086376\\HW0\\error_calc.dat";
    FILE *solution = fopen(solution_file_path, "wt");
    FILE *error_calc = fopen(error_calc_file_path, "wt");
    if (solution == NULL) {
        return 1;
    }
    for (int i = 0; i <= N; i++) {
       fprintf(solution, "%f %f\n", i * h, u[i]);
       fprintf(error_calc, "%f %f %f %f %f\n", i * h, A_dig[i],
           B_dig[i], C_dig[i], d[i]);
    }
    fclose(solution);
    return 0;
}*/
```

```
//int main() {

    void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
        mxArray *prhs[]) {
    // declaring all the parametrs:
    float h, bc_0, bc_N, start, end;
    int ie, is, N;
    char boundary_condition;

    // call Init function
    Init(&N, &boundary_condition, &start, &end, &bc_0, &bc_N);

    // Allocate memory for arrays
    float *u = malloc((N + 1) * sizeof(float));
    float *a = malloc((N + 1) * sizeof(float));
    float *b = malloc((N + 1) * sizeof(float));
    float *c = malloc((N + 1) * sizeof(float));
    float *d = malloc((N + 1) * sizeof(float));
    float *A_dig = malloc((N + 1) * sizeof(float));
    float *B_dig = malloc((N + 1) * sizeof(float));
    float *C_dig = malloc((N + 1) * sizeof(float));
    float *A_dig_er = malloc((N + 1) * sizeof(float));
    float *B_dig_er = malloc((N + 1) * sizeof(float));
    float *C_dig_er = malloc((N + 1) * sizeof(float));
    float *d_er = malloc((N + 1) * sizeof(float));

    // calls all the main functions
    calc_h(&h, start, end, N);
    calc_a(a, N);
    calc_b(b, h, N);
    calc_c(c, h, N);
    RHS(d, h, N);
    LHS(a, b, c, A_dig, B_dig, C_dig, h, N);
    Boundary_conditions(A_dig, C_dig, d, N, h, &is, &ie, u,
        boundary_condition, bc_0, bc_N);
    error_calc(A_dig, B_dig, C_dig, d, A_dig_er, B_dig_er,
        C_dig_er, d_er, N);
    tridiag(A_dig, B_dig, C_dig, d, u, is, ie);
    //output(N, h, u, A_dig, B_dig, C_dig, d); IN A COMMENT
        BECAUSE IM USING MATLAB

   // Prepare output mxArrays
    plhs[0] = mxCreateNumericMatrix(1, N + 1, mxSINGLE_CLASS,
        mxREAL);
```

```
    plhs[1] = mxCreateNumericMatrix(1, N + 1, mxSINGLE_CLASS,
        mxREAL);
    plhs[2] = mxCreateNumericMatrix(1, N + 1, mxSINGLE_CLASS,
        mxREAL);
    plhs[3] = mxCreateNumericMatrix(1, N + 1, mxSINGLE_CLASS,
        mxREAL);
    plhs[4] = mxCreateNumericMatrix(1, N + 1, mxSINGLE_CLASS,
        mxREAL);

    float *output_u = (float *)mxGetData(plhs[0]);
    float *output_A_dig = (float *)mxGetData(plhs[1]);
    float *output_B_dig = (float *)mxGetData(plhs[2]);
    float *output_C_dig = (float *)mxGetData(plhs[3]);
    float *output_d_er = (float *)mxGetData(plhs[4]);

    for (int i = 0; i <= N; i++) {
        output_u[i] = (float)u[i];
        output_A_dig[i] = (float)A_dig_er[i];
        output_B_dig[i] = (float)B_dig_er[i];
        output_C_dig[i] = (float)C_dig_er[i];
        output_d_er[i]  = (float)d_er[i];
    }
    // Free allocated memory
    free(a);
    free(b);
    free(c);
    free(d);
    free(A_dig);
    free(B_dig);
    free(C_dig);
    free(A_dig_er);
    free(B_dig_er);
    free(C_dig_er);
    free(d_er);
    free(u);
}
```

7.2 Matlab script

```
    clc,clear, close all;
%% Input parameters
Boundary_condition = 'Neumann';        % can be either Dirichlet
    or Neumann
```

```
a = 0;                                  % starting x value for the
    calculation
b = 1;                                  % ending x value for the
   calculation
Boundary_condition_initial = 1;         % for Dirichlet it's Y_0
   and for  Neumann its Y'_0
Boundary_condition_final   = -1;        % for Dirichlet it's Y_N
   and for  Neumann its Y'_N
for N = [100 500 1000]
    for precision = {'single','double'}
        %% Writing the input.txt file
        if strcmp(Boundary_condition, 'Neumann')
            bc = 'N';
        elseif strcmp(Boundary_condition, 'Dirichlet')
            bc = 'D';
        end
        ID = 'C:\Users\roiba\Documents\CFD_086376\HW0\input.txt'
            ;
        file = fopen(ID, 'wt');
        fprintf(file, '%d %f %f %c %f %f\n', N, a, b, bc,
            Boundary_condition_initial, Boundary_condition_final)
            ;
        fclose(file);
        x = linspace(a,b,N+1);
        %% Running the C program
        % For new users please run mex -setup in the command
            window and follow the
        % instructions in the Readme file on Github: https://
            github.com/roibaruch24/CFD_086376
        if strcmp(precision, 'single')
            mex cfd1.c
            [u, A_dig, B_dig, C_dig, d] = cfd1();
        elseif strcmp(precision, 'double')
            mex cfd1_double_precision.c
            [u, A_dig, B_dig, C_dig, d] = cfd1_double_precision
                ();
        end
        %% Ploting the function
        figure(1)
        plot(x', u', 'DisplayName', ['N=' num2str(N) ',
            Precision:' char(precision)]);
        hold on;
        legend ('show','Interpreter', 'latex')
        title("y(x)")
        subtitle(['Boundary condition are:' Boundary_condition
```

```
    ])
xlabel("x")
ylabel("y")
grid on
%% Error calculation
N_vec=linspace(1,N,N+1);
h=(b-a)/N;
switch Boundary_condition
    case 'Dirichlet'
        low_dig = diag(A_dig(3:N-1),-1);
        mid_dig = diag(B_dig(2:N-1));
        high_dig = diag(C_dig(2:N-2),1);
        LHS  = low_dig + mid_dig + high_dig;
        figure(2);
        error =  LHS*u(2:N-1)'-d(2:N-1)';
        semilogy(h*N_vec(1:N-3), abs(error(1:end-1)), '
            DisplayName', ['N=' num2str(N) ',Precision
            :' char(precision)], 'LineWidth', 1);
        legend show
        hold on;
        title("Error on a logarithmic scale")
        subtitle(['Boundary condition are:'
            Boundary_condition])
        xlabel("X")
        ylabel("Error")
        grid on;
    case 'Neumann'
        low_dig = diag(A_dig(2:N+1),-1);
        mid_dig = diag(B_dig(1:N+1));
        high_dig = diag(C_dig(1:N),1);
        LHS  = low_dig + mid_dig + high_dig;
        figure(3);
        error = LHS*u(1:N+1)'-d(1:N+1)';
        semilogy(x, abs(error(1:end)), 'DisplayName', ['
            N=' num2str(N) ',Precision:' char(
            precision)], 'LineWidth', 1);
        legend show
        hold on;
        title("Error on a logarithmic scale")
        subtitle(['Boundary condition are:'
            Boundary_condition])
        xlabel("X")
        ylabel("Error")
        grid on;
end
```

```
    end
end
%% boundary conditions
switch Boundary_condition
    case 'Neumann'
        figure(1);
        bc_start = Boundary_condition_initial*x(1:N/3)+(u(1)-
            Boundary_condition_initial*x(1));
        bc_end = Boundary_condition_final*x(end-N/3:end)+(u(N)-
            Boundary_condition_final*x(N));
        plot(x(1:N/3),bc_start,'--','DisplayName','$y''(0)=1$'
            )
        plot(x(end-N/3:end),bc_end,'--','DisplayName','$y''(1)=
            -1$')
        legend ('show','Interpreter', 'latex')
end
```