Programming Assignment #2 : Solving 1D Wave Equation

Jerin Roberts
November 16, 2016

Supervisor: Dr. Carl Ollivier-Gooch
Locations: University of British Columbia

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Overview

The are many physical phenomenons in physics and engineering that require linear and non-linear partial differential equations to described the true nature of the system. Solving these systems analytically and finding exact solutions for these equations can be difficult and often require simplifications that ultimately don't fully represent the problem being investigated. Numerical methods for solving PDE's provides a means of finding approximations to the exact solutions without having to make sacrificial simplifications. With recent advancements in computational technology numerical methods can now be easily applied to large and difficult problems that would otherwise be impossible to solve.

1.1. Theory

Numerical problems are essentially solved by breaking the entire solution domain into small discrete points (mesh) and finding the solution at or around these areas. Each point requires the solving the differential equations that represent physical phenomenon being investigated. Since the exact solution cannot be computed, it is instead approximated using various techniques and methods.

The Runge-Kutta scheme is a family of numerical techniques used to solve ordinary differential equations by numerically integrating using trial steps at the midpoints of an interval to cancel out lower-order error terms. A two stage Runge-Kutta scheme is displayed in equation 1.2 which uses Explicit Euler as the first stage intermediate step 1.1.

$$w^{(1)} = w^n + \triangle t(\lambda w^n) \tag{1.1}$$

$$w^{n+1} = w^n + \triangle t\left(\frac{\lambda w^n + \lambda w^{(1)}}{2}\right) \tag{1.2}$$

The program will implement a 2nd order up wind flux evaluation method and a two-stage Runge-Kutta time advance scheme. For each interior volume cell the net flux will be calculated for a specific time step using the second order up wind method. The flux for the face $i = 3/2$ is displayed in equation 1.4. The flux is evaluated at each face and then the faces surrounding a single cell are summed to find the flux through that cell. Integrating this into equation 1.3 the wave equation can be evaluated using equation 1.5

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} = 0 \tag{1.3}$$

$$T_{\frac{3}{2}} = \frac{3\overline{T}_1 - \overline{T}_0}{2} \tag{1.4}$$

$$\frac{\partial T}{\partial t} + u\left(\frac{3\overline{T}_i - 4\overline{T}_{i-1} + \overline{T}_{i-2}}{2\triangle x}\right) = 0 \tag{1.5}$$

Boundary conditions will be implement using ghost cells allowing the interior scheme to remain the same during calculation of bordering cells. Because the interior scheme requires

two upwind values, two ghost cells are required for this scheme to run on the first interior cell $i = 1$. The ghost cells are calculated such the boundary condition is enforced at $i = 1/2$.

$$\overline{T}_0 = 2\overline{T}_w - \overline{T}_1 \tag{1.6}$$

$$\overline{T}_{-1} = -2\overline{T}_w + 3\overline{T}_0 \tag{1.7}$$

The value for the ghost cells implementing Neumann Boundary at $x = 0$ is calculated using equation 1.6 for cell $i = 0$ and 1.7 for cell $i = -1$.

2. IMPLEMENTATION

2.1. PROGRAM OVERVIEW

The C/C++language was selected for this programming assignment. The scripts were compiled using g++/gcc version 5.4.0 on Ubuntu 16.04.02 and are available in the attached zip or for clone via the link provided: `https://github.com/j16out/cfd510` . The program itself is broken into 3 pieces and two levels to produce a modular set that makes it easier to apply to different problems. The highest level contains the "macro" or the main function which can be modified for different problems. The numerical directory contains numerical.cpp script and its header file numerical.hpp. This set contains all the functions necessary for solving the problem numerically. Table 2.1 displays all functions with a short description of each. The vroot directory contains the scripts necessary for drawing data. These scripts make use of the ROOT-v6 libraries. ROOT is a popular data analysis framework primarily written in C++ and Phython. For more information on ROOT libraries visit the link provided: `https://root.cern.ch/` .

Function	Purpose
set_ array_ size()	Set size of array if not default(160x160)
set_ ghostcells()	Set Ghost cells and update boundary conditions
set_ zero()	Zero entire array including ghost cells
print_ array()	Prints array in terminal style output
gs_ iter_ SOR()	Performs single Gauss-Seidel iteration with over-relaxation, over first in y then x
calc_ source()	Calculates the source term for the problem
calc_ newcell()	Calculates new solution at single grid point
get_ surcells()	Gets current solution of neighboring cells
solve_ arraySOR()	Solves for solution by performing many GS-iterations until defined convergence is met
get_ discrete_ Error()	Returns error for solution at P(1/2,1/2) for Poisson Problem
get_ solution()	Returns Solution at P(1/2,1/2) for Poisson Problem via averaging scheme
get_ l2norm()	Returns L^2 norm error between two arrays of equal size
set_ analytic()	Set array values to predefined exact solution

Table 2.1: List of Program Functions

The functions act on a structure called $carray$ which contains the solution domain array and its various parameters. The Struct contains the main array, its defined working area (mesh size), data storage vectors, iteration count, and the represented dimension between points. Having these organized in a struct provides a compact way of passing and modifying the array and all its pertinent parameters. The outline of the struct used for the wave equation problem is shown below.

```
struct carray{
//arrays
float mcellSOL [maxx][maxy];//first stage and solution mesh
float mcellSOL2 [maxx][maxy];//second stage solution mesh
float mcellFI [maxx][maxy];//first stage flux
float mcellFI2 [maxx][maxy];//second stage flux

//array attributes
int sizex = maxx;
int sizey = maxy;
float DIM1 = 0;

//current time
float tstep = 0;
float ctime = 0;

//temporary cells to store
```

```
float Tim1_j=0.0;
float Tim2_j=0.0;
float Ti_j=0.0;

};
```

2.2. 1D Wave Problem

The program was used to solve and find solutions to the 1D wave equation with information moving left to right (2.1) with satisfying conditions (2.2). An exact solution to this problem is provided and written as $T(x,t)$ and displayed in equation 2.3 which will enabled a comparison for solution error assessment.

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} = 0 \tag{2.1}$$

$$0 \leq x \leq x_{max} \qquad 0 \leq t \qquad u = 2 \tag{2.2}$$

$$T(x,t) = \sin(2\pi(2t - x)) \tag{2.3}$$

2.3. Boundary Conditions

Boundary Conditions are implemented through the calculation of the ghost cells. For the purely up wind scheme, we require two ghost cells to be calculated. The ghost cell is calculated such that the boundary is implemented at the face between the first ghost cell and first border cell. Below shows the function that implements boundaries for the Wave problem.

```
void set_ghostcells(carray & myarray)
{
//set boundary conditions in ghost cells
myarray.mcellSOL2[0][1] = -2.0*(sin(4.0*PI*myarray.ctime)) +
    3.0*myarray.mcellSOL[1][1];
myarray.mcellSOL2[1][1] = 2.0*(sin(4.0*PI*myarray.ctime)) -
    myarray.mcellSOL[2][1];
}
```

A loop is not necessary as this is purely a 1D problem therefore only two cells need to be updated each time step iteration. This subroutine will need to be modified when the program is applied to different problems or using different flux schemes at the boundary. At this stage is not changeable during run-time. Initial Conditions are set before the problem is solved using the subroutine shown below.

```
void set_intial_cond(carray & myarray)
{
float DIM1 = myarray.DIM1;
float dx =0.0;
```

```
float f;
for(int j = 1; j < myarray.sizey-1; ++j)
{
    for(int i = 2; i < myarray.sizex; ++i)
    {
    dx = (i-1.5)*DIM1;
    f = -sin(2.0*PI*dx);
    myarray.mcellSOL[i][j] = f;
    }
}
}
```

The loop calculates all the values of T at time t=0 for varying x using equation 2.4. Similar to the boundary conditions the initial conditions are not modifiable during run-time and will need to be tailored to specific problems.

$$T(x,0) = -\sin(2\pi x) \tag{2.4}$$

2.4. Iteration Scheme

A RK2 routine was implemented to solve the system of equations approximating the solution at each grid point which is denoted as function $solve\_arrayRK2$. The routine starts from the first interior cell $i = 2$ and iterates flux integration schemes until $i = i_{max}$. Once the flux values have been gathered for each face and expressed as a single cell value a subroutine calculates the time advance solution at each cell. Below shows both the flux calculating and solution calculating subroutines respectively.

```
void get_FIarray(carray & myarray, int stage)
{
for(int j = 1; j < myarray.sizey-1; ++j)
{
    for(int i = 3; i < myarray.sizex; ++i)
    {
    //----get surrounding cells and compute new cell-------//
    get_surcells(myarray, i, j, stage);
    float newcell = calc_2nd_UW(myarray);
    //-----update current cell----//
    if(stage == 1)
    myarray.mcellFI[i][j] = newcell;
    if(stage == 2)
    myarray.mcellFI2[i][j] = newcell;
    }
}
}
```

```
void get_RK2(carray & myarray, int stage)
{
```

```
if(stage == 1)
{
for(int j = 1; j < myarray.sizey-1; ++j)
{
    for(int i = 2; i < myarray.sizex; ++i)
    {
      myarray.mcellSOL2[i][j] =
          myarray.mcellSOL[i][j]-myarray.tstep*(myarray.mcellFI[i][j]);
    }
}
}
if(stage == 2)
{
for(int j = 1; j < myarray.sizey-1; ++j)
{
    for(int i = 2; i < myarray.sizex; ++i)
    {
     myarray.mcellSOL2[i][j] =
         myarray.mcellSOL[i][j]-myarray.tstep*((myarray.mcellFI2[i][j]+myarray.mcellFI[i][j])/2.0)
    }
}
}
}
```

It should be noted the first interior cell i = 2 flux is calculated in a seperate sub routine called $get\_FIarray\_1stcell$. An outer solving loop implements the RK2 scheme via two stages for all required time steps. After the two stages are computed the scheme is advanced by one time step and the boundary conditions are re-evaluated and implemented by calculating the new ghost cell values for the latest solution. Below displays the solution solving loop:

```
void solve_arrayRK2(carray & myarray, float tmax, float cfl)
{
float tstep = (cfl*(myarray.DIM1))/2.0;
myarray.tstep = tstep;
float ctime = myarray.ctime;

//set intial conditions
set_intial_cond(myarray);
set_ghostcells(myarray);

  while(ctime < tmax-tstep)
  {

   for(int h = 1; h <= 2; ++h)
    {
    get_FIarray_1stcell(myarray, h);//(array, stage)
    get_FIarray(myarray, h);//(array, stage)
    get_RK2(myarray, h);//(array, stage)
    }
```

```
  //mv sol2 back to array sol1
  mv_SOL2_to_SOL1(myarray);

  //flux at boundary
  set_ghostcells(myarray);//evaluates to ghost cells for array

  myarray.ctime = myarray.ctime+myarray.tstep;
  ctime = myarray.ctime;
  ++n;
  }
}
```

The scheme iterates until the required end time is reached. The time steps are determined by the CFL number and the mesh size. Once the solution has been solved a analytical solution is produced and used for an error comparison.

3. RESULTS

3.1. VALIDATION

While implementing the boundaries as stated above the solution was computed at $t = 1$ using a CFL number of 0.4. This will enable the wave to be propagated twice across the domain. The solution was then plotted in figure 3.1 for meshes of refining size in comparison with the analytical solution.

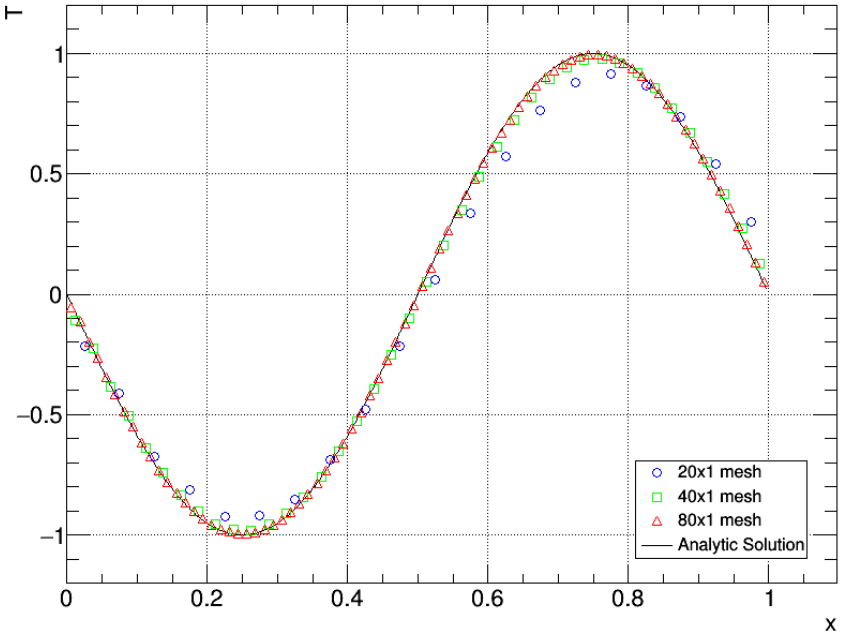


Figure 3.1: The solution at $t = 1$ for mesh sizes of 20, 40, and 80

The order of accuracy for the scheme was determined using the log log method as displayed in figure 3.2. The order of accuracy for this scheme was estimated to be 2nd order accurate based on the L_2 data from 6 different meshes. This matches what was expect by implementing a 2nd order interior and boundary flux evaluation scheme.

Mesh Size	L^2**norm**	$\triangle x$	$\triangle t$	**Order**
20 x 1	$8.158*10^{-2}$	0.05	0.01	-
40 x 1	$2.399*10^{-2}$	0.025	0.005	1.934
80 x 1	$6.445*10^{-3}$	0.0125	0.0025	1.937
160 x 1	$1.676*10^{-3}$	0.00625	0.00125	1.942

Table 3.1: Table of L^2 norms for increasing mesh size for $CFL = 0.4$

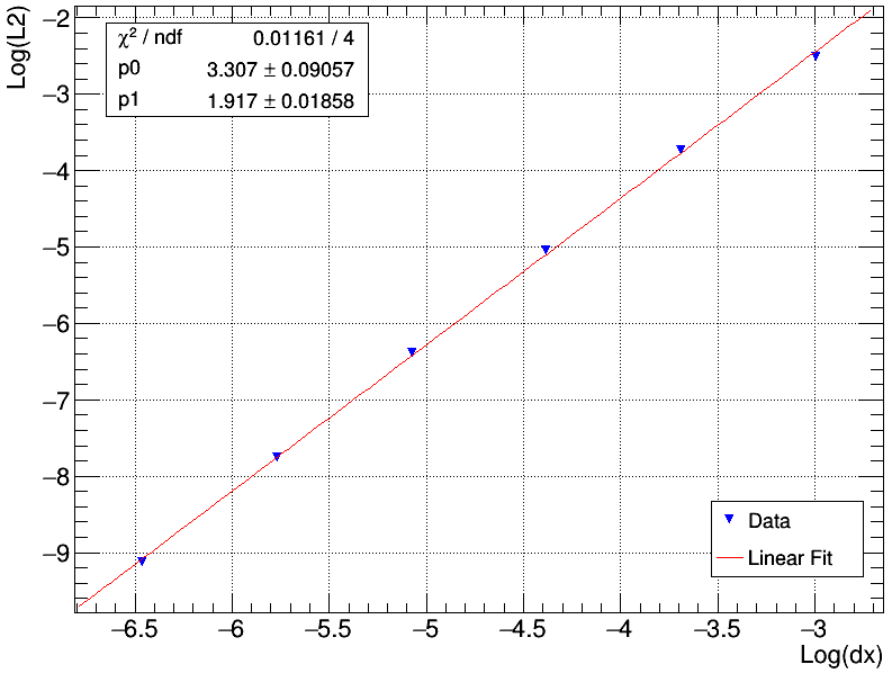


Figure 3.2: Calculated Order of Accuracy

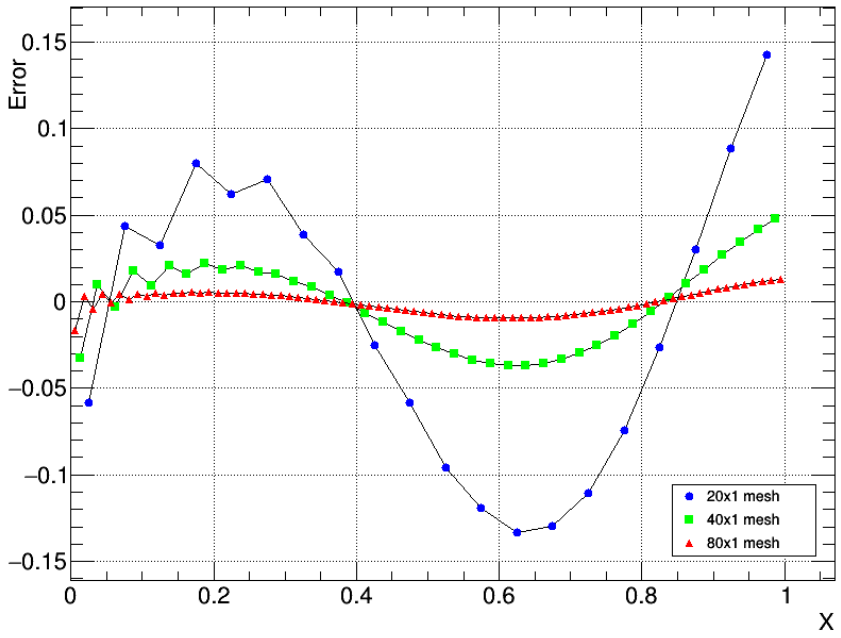


Figure 3.3: Solution Error at $t = 1$ for mesh sizes of 20, 40, and 80

The log-log plot enables us to estimate what mesh size would be required for a particular error. Using the plot it was found in order to obtain an error less than 10^{-3} a mesh size of 206.12 or greater is required. Similarly to obtain an error less than 10^{-4} a mesh of greater than 685.14 is required using a CFL number of 0.4.

3.2. STABILITY

Stability is an important factor when considering time advance schemes. Ideally one would like to advance to the required time solution is as few time steps as possible to reduce computation load. However if the time step is too large instabilities can arise causing the solution error to diverge exponentially. These can arise from the time dependent boundary, with a large time step the next solution change is calculate from a large boundary change which results in a large solution overshoot. The problem compounds with the large overshoot as again the solution and time step boundary are vastly different. The stability limits of the program were experimentally determined by evaluating the L2 norm as a function of increasing CFL number. A small CFL number corresponds to a small time step, therefore the L2 norm was calculated relatively against an ideal stable time step.

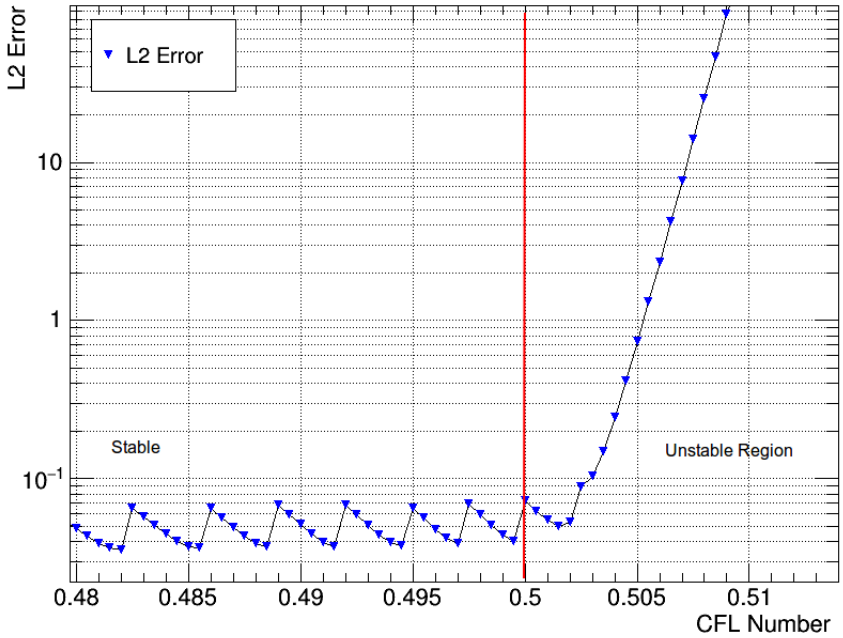
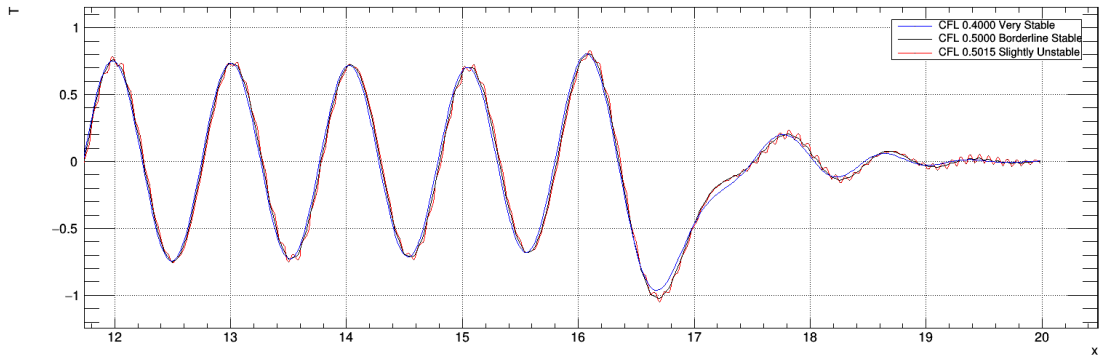


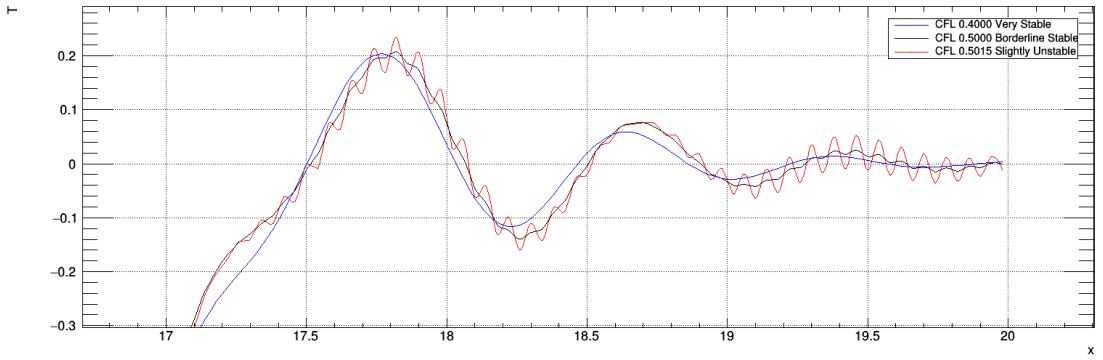
Figure 3.4: The L2 error for increasing CFL number for a 500x1 mesh with $x_{max} = 20$

The domain was extended to $x_{max} = 20$ with the first section ($0 \leq x \leq 1$) is set as $-x$ and 0 everywhere else as initial conditions. The solution was computed at t=8 on a mesh of 500 control volumes. Figure 3.4 displays the L2 norm error as a function of CFL near the instability regime. The plot identifies when the scheme becomes violently unstable in which case the solution is not a good representation of the exact solution. The maximum stable CFL was

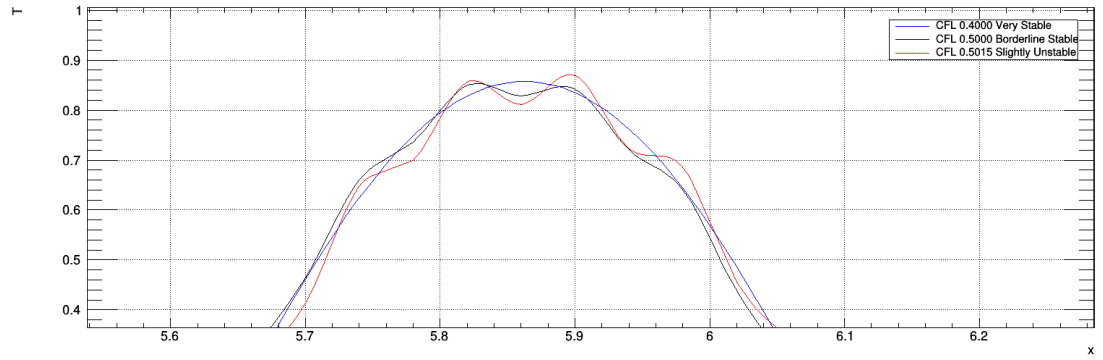
determined to be 0.500 ± 0.005. Anything above the value is considered slightly unstable or completely unstable.



(a)



(b)



(c)

Figure 3.5: Comparison between a stable(blue), borderline (black) stable and slightly unstable (red) solution at $t = 8$ for 500 control volumes

Figure 3.5 displays the visible differences between a stable, borderline stable and slightly unstable solution. Figure 3.5 part a) displays a zoomed out version of the solution. Only

the slightly unstable solution has a visible difference. The instability is seen as the higher frequency oscillations developing about the path of the solution. As we zoom in further we can begin to see the board-line stable solution which appeared stable also contains some oscillatory characteristics in zoomed regions. These features grow exponentially with a further increase in CFL. It should be noted these features begin to appear at the same time the error begins to diverge, providing a simple means of determining the stability boundary of the solution. From theory we see the stability region is dependent on both the stability bounds of the 2nd order upwind method and the RK2 method. The 2nd upwind stability region is a circle with radius of $\frac{2u\triangle t}{\triangle x}$, while the RK2 is denoted as a approx circle of radius 1 (on real axis). Setting these equal and rearranging we find the theoretical CFL to be 0.5 therefore CFL values between 0 and 0.5 should be stable. Comparing Experimental and theoretical values of max CFL for determining stability we find they are indeed consistent.

3.3. Effect of Boundary Conditions

Initial and boundary conditions are the parameters that differentiate a problem from its general form. Failure to setup or apply appropriate boundary conditions means your solving a different problem which is going to give you the wrong solution. In addition to applying the correct boundary there are different methods for implying the same boundary, these methods however are not all created equal. The trade off is usually split between implementation and error mitigation. Complicated schemes usually reduce error, but sometimes can be at the price of performance or maintenance. Boundary condition methods should be tailored to the problem at hand.

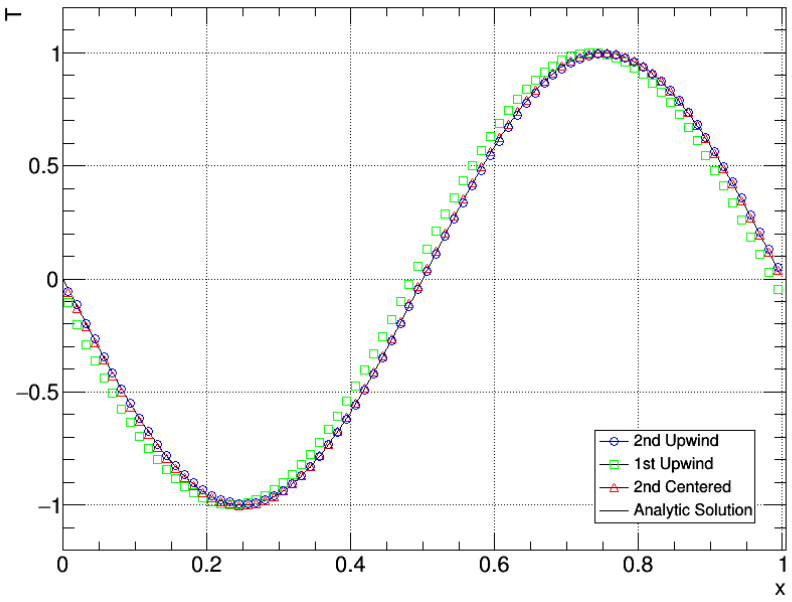


Figure 3.6: The maximum change in solution for $\omega = 1, 1.3, 1.5$ on a 20 x 20 mesh

For the wave equation three types of boundaries were implemented and compared to highlight their strengths and weaknesses. In addition to the 2nd order upwind boundary implementation used in the previous sections a 1st order upwind and 2nd centered scheme were investigated for mesh of 80x1 for a CFL of 0.4 where $x_{max} = 1.0$. It should be noted the interior scheme remains 2nd order upwind. The solution implementing each boundary condition is plotted in 3.6 and the relative error against the exact solution in 3.7.

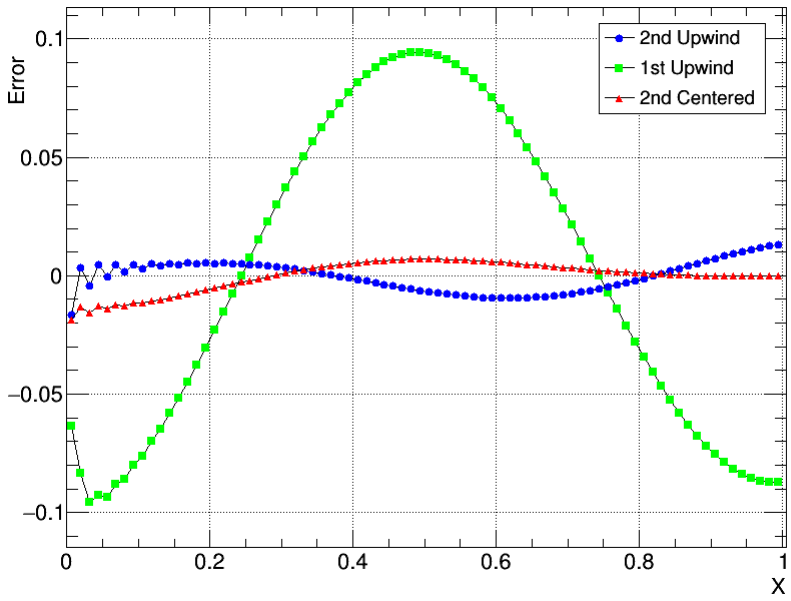


Figure 3.7: The maximum change in solution for $\omega = 1, 1.3, 1.5$ on a 20 x 20 mesh

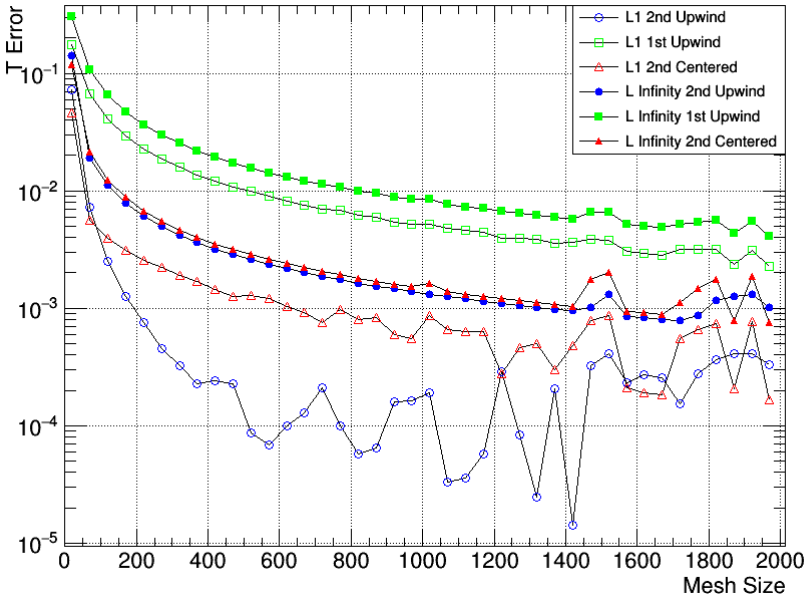


Figure 3.8: The maximum change in solution for $\omega = 1, 1.3, 1.5$ on a 20 x 20 mesh

4. Conclusion

This project provides great inside to the internal algorithms used for calculating numerical solutions for symmetric grids. The Laplace problem provided a great platform for developing and testing the program. Having the analytic solution really help fine tune the program and helped given an idea on how accurate the solutions could be. Applying the program to Poisson problem showed how one could find a solution and still estimate error without a known solution to compare with. Numerical methods provide a great way of solving difficult problems and until new methods are discovered for finding exact solution will remain the main method for finding solutions to difficult problems.

A. Appendix

A.1. Poisson.cpp

```
/*-------------------------------------------------------------------------------//
Main Program for finding pressure for imcompressible flows using Poisson
    equations.
Finds solution at P(1/2,1/2) for Land descretization error for w values of 1
    for
20x20,40x40 and 60x60 array

Jerin Roberts 2016
```

```
compiled using g++/gcc version 5.4.0 on Ubuntu 16.04.02 and are available for
    clone
via the link provided: url{https://github.com/j16out/
//-------------------------------------------------------------------------------*/


#include <vector>
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <sstream>
#include <math.h>
#include "TApplication.h"
#include "vroot/root.hpp"
#include "numerical/numerical.hpp"

using namespace std;


#define E07 0.0000001
#define E08 0.00000001
#define E09 0.000000001
#define E10 0.0000000001
#define E11 0.00000000001




int main(int argc, char **argv)
{


carray poisson1;//my main array
carray poisson2;
carray poisson3;

//set array size or default used 162x162
set_array_size(poisson1, 20, 20, 1.0);//array, xsize, ysize, dimension
set_array_size(poisson2, 40, 40, 1.0);
set_array_size(poisson3, 80, 80, 1.0);


//set ghost cells as boundary conditions


set_zero(poisson1);
```

```
set_ghostcells(poisson1);//set ghost cells/boundaries
//print_array(poisson1);//print array in terminal


set_zero(poisson2);
set_ghostcells(poisson2);
//print_array(poisson2);

set_zero(poisson3);
set_ghostcells(poisson3);
//print_array(poisson3);



//---------------------GS SOR w=1.3 loop 1----------------------//

solve_arraySOR(poisson1, E11, 1.3);
cout << "Solution: " << get_solution(poisson1) << "\n";


//---------------------GS SOR w=1.3 loop 2----------------------//

solve_arraySOR(poisson2, E11, 1.3);
cout << "Solution: " << get_solution(poisson2) << "\n";

//---------------------GS SOR w=1 loop 3----------------------//

solve_arraySOR(poisson3, E11, 1.3);
cout << "Solution: " << get_solution(poisson3) << "\n";

//---------------------calc error based on ASME---------------//

get_discrete_Error(poisson3, poisson2, poisson1, 1.0);


//----------------------Draw Data---------------------//



if(1)//start root application
{
   TApplication theApp("App", &argc, argv);
   draw_3DgraphP(poisson3);//draw 3d graph
   theApp.Run();
}



//end
```

```
}
```

A.2. NUMERICAL.HPP

```
#ifndef numerical_INCLUDED
#define numerical_INCLUDED


#include <vector>
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <sstream>
#include <math.h>
#include <iomanip>

using namespace std;

#define BIG 1000000
#define maxx 160
#define maxy 160
#define PI 3.141592654



struct carray{
float mcell [maxx][maxy];
vector<float> l2norm;
vector<float> diff;
int sizex = maxx;
int sizey = maxy;
int iterations = 0;
float DIM1 = 0;

};


void set_array_size(carray & myarray, int x, int y, float DIM);//set array size

void set_ghostcells(carray & myarray);//set ghost cells


void set_zero(carray & myarray);//zero entire array
```

```
void print_array(carray & myarray);//print array in terminal

float gs_iter_SOR(carray & myarray, float omega);//complete one gauss-seidel
    iteration with SOR

float calc_source(carray & myarray, int i, int j);//calculate source term

float calc_newcell(carray & myarray, float source, float Tip1_j, float Tim1_j,
    float Ti_jp1 ,float Ti_jm1);//calculate new cell value based on 2nd order
    scheme

void get_surcells(carray & myarray, float & Tip1_j, float & Tim1_j, float &
    Ti_jp1 ,float & Ti_jm1, int i, int j);//obtain values of surrounding cells

void solve_arraySOR(carray & myarray, float E0, float w);//solve the array
    using gs-iterations

void get_discrete_Error(carray ray1, carray ray2, carray ray3, float
    DIM);//get error using 3 arrays based on ASME solution accuarcy handout

float get_solution(carray & myarray);//find solution at p(1/2,1/2) for Poisson

float get_l2norm(carray & myarray, carray myarray2);//get estimated vale for
    l2 norm between arrays


#endif
```

A.3. NUMERICAL.CPP

```
#include "numerical.hpp"




//----------set array size (working area excluding ghost)---------------//

void set_array_size(carray & myarray, int x, int y, float DIM)
{
   if(x < 160 && y < 160)
   {
   myarray.sizex = x+2;
   myarray.sizey = y+2;
   myarray.DIM1 = DIM/(x);
   }
   else
```

```
   cout << "Array size to big, setting to default 160" << "\n";

}


//--------------------------zero array----------------------------//

void set_zero(carray & myarray)
{
   for(int i = 0; i < myarray.sizex; ++i)
   {
      for(int j = 0; j < myarray.sizey; ++j)
      {
      myarray.mcell[i][j] = 0;//set everything to zero

      }
   }
}


//--------------------------set ghost cells for
    Poisson-----------------------------//


void set_ghostcells(carray & myarray)
{
float DIM1 = myarray.DIM1;

//set boundary conditions in ghost cells
for(int i = 1; i < myarray.sizex-1; ++i)
   {float d = (i-0.5)*DIM1;

   //neumann boundaries
   myarray.mcell[i][0] = myarray.mcell[i][1];//top ghost cells
   myarray.mcell[0][i] = myarray.mcell[1][i];//left ghost cells

   //dirichlet boundaries
   myarray.mcell[i][myarray.sizex-1] = 2.0*(5.0-((1.0/2.0)*pow((1.0+pow(d,
       2)),3))) - myarray.mcell[i][myarray.sizex-2];
   myarray.mcell[myarray.sizey-1][i] = 2.0*(5.0-((1.0/2.0)*pow((1.0+pow(d,
       2)),3))) - myarray.mcell[myarray.sizex-2][i];

       }
}



//--------------------------Guass-Seidel SOR----------------------------//
```

```
float gs_iter_SOR(carray & myarray, float omega)
{
float DIM1 = myarray.DIM1;//get dimensions of array (not grid size)
float Tip1_j, Tim1_j, Ti_jp1, Ti_jm1;//define values for surrounding cells
float l2sum = 0.0;
float sx = myarray.sizex-2;
float sy = myarray.sizey-2;

float dx = 0.0;//change in x
float dy = 0.0;//change in y

carray oldarray=myarray;


//-----iterate through all x for steps y -----//
set_ghostcells(myarray);

for(int j = 1; j < myarray.sizey-1; ++j)
{


   for(int i = 1; i < myarray.sizex-1; ++i)
   {

   dx = (i-0.5)*DIM1;
   dy = (j-0.5)*DIM1;

   //----get surrounding cells and compute new cell-------//
   get_surcells(myarray, Tip1_j, Tim1_j, Ti_jp1 , Ti_jm1, i, j);
   float source = calc_source(myarray, i, j);
   float newcell = calc_newcell(myarray, source, Tip1_j, Tim1_j, Ti_jp1 ,
       Ti_jm1);

   //----apply over-relaxation-----//
   float delta = newcell - myarray.mcell[i][j];
   float newcellSOR = myarray.mcell[i][j] + omega*(delta);

   //-----update current cell----//
   myarray.mcell[i][j] = newcellSOR;


   }

}

set_ghostcells(myarray);
//-----iterate through all y for steps x -----//
for(int i = 1; i < myarray.sizey-1; ++i)
{
```

```
   for(int j = 1; j < myarray.sizex-1; ++j)
   {

   dx = (i-0.5)*DIM1;
   dy = (j-0.5)*DIM1;

   //----get surrounding cells and compute new cell-------//
   get_surcells(myarray, Tip1_j, Tim1_j, Ti_jp1 , Ti_jm1, i, j);
   float source = calc_source(myarray, i, j);
   float newcell = calc_newcell(myarray, source, Tip1_j, Tim1_j, Ti_jp1 ,
       Ti_jm1);

   //----apply over-relaxation-----//
   float delta = newcell - myarray.mcell[i][j];
   float newcellSOR = myarray.mcell[i][j] + omega*(delta);


   //-----update current cell----//
   myarray.mcell[i][j] = newcellSOR;

   }

}


float maxdiff = -1.0;

for(int i = 2; i < myarray.sizey-2; ++i)
{
   for(int j = 2; j < myarray.sizex-2; ++j)
   {
   float diff = abs(oldarray.mcell[i][j] - myarray.mcell[i][j]);
      if(diff > maxdiff)
      {
      maxdiff = diff;
      //cout << "coord " << maxdiff << " " << i << " " << j << "\n";
      }
   }

}

//for obtaining l2norm convergence

/*carray analytic;
set_array_size(analytic, 10, 10, 1.0);
set_analytic(analytic);
//float norm = get_l2norm(myarray, analytic);
```

```
myarray.l2norm.push_back(norm); */
myarray.diff.push_back(maxdiff);
++myarray.iterations;

return maxdiff;
}
//-------------------------Get L2 nrom for unknown
    analytical----------------------//

float get_l2norm(carray & myarray, carray myarray2)
{
float l2sum =0;
float sx = myarray.sizex-2;
float sy = myarray.sizey-2;

for(int j = 1; j < myarray.sizey-1; ++j)
{
   for(int i = 1; i < myarray.sizex-1; ++i)
   {

   float P = myarray.mcell[i][j];
   float T = myarray2.mcell[i][j];
   l2sum = l2sum + pow((P-T),2);

   }

}

float l2 = sqrt(l2sum/(sx*sy));
cout << "L2 norm: " << l2 << "\n";
return l2;
}

//--------------------------Solve array using
    GS-iterations-----------------------------//

void solve_arraySOR(carray & myarray, float E0, float w)
{
printf("\n\nSolving Grid size: %d Relaxation: %f\n", myarray.sizex, w);
bool relax_on = true;
float diff = 1;// current difference
float ldiff = BIG;// previous difference
int div = 0;
int update = 0;
int update2 = 100;

while(diff >= E0)
{
```

```
diff = gs_iter_SOR(myarray, w);


   if(diff > BIG)//avoid infinite loops if diverges
   break;

   if(update >= update2)//report difference every 100 steps
   {cout << "Update: step " << update << " Solution Change: " <<
       setprecision(9) << fixed << diff << " \n";
   //print_array(myarray);
    update2 = update2 + 100;
   }

   if(ldiff == diff)//checks for repeated values indication of instability for
       high w
   ++div;
   else
   div = 0;

   if(div > 3 && w > 1.1)//reduces over-relaxation for high w when unstable
   {
   w = 1.0;
   cout << "Relaxation Reduced to "<<w<<" @ " << myarray.iterations << " \n";
   }

ldiff = diff;
++update;
}

cout << "Iterations: " << myarray.iterations << "\n";
}


//--------------------------Print array in
    terminal-----------------------------//

void print_array(carray & myarray)
{
cout << "\n";

   for(int j = 0; j < myarray.sizey; ++j)
   {
   cout << "\n|";
      for(int i = 0; i < myarray.sizex; ++i)
      {
      if(myarray.mcell[i][j] >= 0)
      cout << setprecision(3) << fixed << myarray.mcell[i][j] <<"|";
      if(myarray.mcell[i][j] < 0)
      cout << setprecision(2) << fixed << myarray.mcell[i][j] <<"|";
```

```
     }

   }
cout << "\n";
}

//--------------------Calculate new cell value from neighbors
    ---------------------//


float calc_newcell(carray & myarray, float source, float Tip1_j, float Tim1_j,
    float Ti_jp1 ,float Ti_jm1)
{
float DIM1 = myarray.DIM1;
float chx = DIM1;
float chy = DIM1;
float temp = (pow(chx,2)*pow(chy,2)) / (2*(pow(chx,2)+pow(chy,2)));
float newcell = (( ((Tip1_j+Tim1_j)/pow(chx,2)) + ((Ti_jp1+Ti_jm1)/pow(chy,2))
    - source ) * temp) ;

return newcell;
}
//---------------------Get source term for poisson
    problem----------------------//
float calc_source(carray & myarray, int i, int j)
{
float DIM1 = myarray.DIM1;
float dx = (i-0.5)*DIM1;
float dy = (j-0.5)*DIM1;
float source =
    -1.0*(pow(3*pow(dx,2)-3*pow(dy,2),2)+72.0*(pow(dx,2)*pow(dy,2))+pow(3*pow(dy,2)-3.0*pow(dx,2)
//source = 0 for Laplace problem

return source;
}

//----------------------Get average solution at point
    (1/2)(1/2)--------------------//

float get_solution(carray & myarray)
{
float DIM1 = myarray.DIM1;
int sx = (myarray.sizex)/2.0;
int sy = (myarray.sizey)/2.0;
float sol =
    (myarray.mcell[sx-1][sy]+myarray.mcell[sx][sy]+myarray.mcell[sx][sy-1]+myarray.mcell[sx-1][sy

printf("cell 1: %f cell 2: %f cell 3: %f cell 4:
    %f\n",myarray.mcell[sx-1][sy],myarray.mcell[sx][sy],myarray.mcell[sx][sy-1],myarray.mcell[sx-
```

```
//for Poisson problem only, finds value based on average of four surrounding
    cells

return sol;
}

//-------------------------Get current cell
    values----------------------------//

void get_surcells(carray & myarray, float & Tip1_j, float & Tim1_j, float &
    Ti_jp1 ,float & Ti_jm1, int i, int j)
{
float fcon = false;
float sizex = myarray.sizex;
float sizey = myarray.sizey;

      Tip1_j = myarray.mcell[i+1][j];
      Tim1_j = myarray.mcell[i-1][j];
      Ti_jp1 = myarray.mcell[i][j+1];
      Ti_jm1 = myarray.mcell[i][j-1];

}

//-------------------------Get descrete error----------------------------//

void get_discrete_Error(carray ray1, carray ray2, carray ray3, float DIM)
{
//Calculating error as described in paper "procedure for estimation and
    reporting of uncertainty due to discretization in CFD applications"//

printf("\nCalculating Error...\n");

float h1 = DIM/ray1.sizex;
float h2 = DIM/ray2.sizex;
float h3 = DIM/ray3.sizex;


float sol1 = get_solution(ray1);
float sol2 = get_solution(ray2);
float sol3 = get_solution(ray3);



printf("h1: %f \nh2: %f \nh3: %f, \nsol1: %f \nsol2: %f \nsol3: %f\n",h1, h2,
    h3, sol1, sol2, sol3);

float r21 = h2/h1;
float r32 = h3/h2;
```

```
printf("\nr32: %f \nr21: %f\n",r32, r21);

float e32 = sol3-sol2;
float e21 = sol2-sol1;

float s = (e32/e21);
if(s >= 0)
s = 1;
else
s = -1;

float p_n = 0;
float p = (1/log(r21))*(abs(log(abs(e32/e21))+0));

printf("intial guess: %f \n", p);

float diff = 1;

   while(diff > 0.0000001)
   {

   float p_n =
       (1/log(r21))*(abs(log(abs(e32/e21))+log((pow(r21,p)-s)/(pow(r32,p)-s))
       ));
   diff = abs(p_n -p);
   //printf("p_n: %f p: %f diff: %f\n",p_n, p, diff);

   p = p_n;
   }

//
float sol_ext21 = (pow(r21, p)*sol1-sol2)/(pow(r21,p)-1.0);
float sol_ext32 = (pow(r32, p)*sol2-sol3)/(pow(r32,p)-1.0);

printf("order: %f \nphi_ext21: %f \nphi_ext32 %f\n",p, sol_ext21, sol_ext32);

float ea21 = abs((sol1-sol2)/sol1);

float e_ext21 = abs((sol_ext21-sol1)/sol_ext21);

float GCI_21 = (1.25*ea21)/(pow(r21,p)-1.0);


printf("ea21: %f \ne_ext21: %f \nGC121 %f \n", ea21, e_ext21, GCI_21);

}
```

REFERENCES

[Celik, 2006] Ismail B. Celik1, Urmila Ghia, Patrick J.Roache and Christopher J. Freitas "Procedure for Esitmation and Reporting Uncertainty Due to Discretization in CFD apllications", West Virginia University, Morgantown WV, USA