Programming Assignment #1 : Solving Laplace and Poisson's Equation

Jerin Roberts

October 31, 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 descrete 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 most common methods are finite difference, finite element, and finite volume. A program was created to numerically solve Laplace and Poisson steady state problems. The program will implement a 2nd order finite volume method. For each volume cell the net flux will be calculated and used to evaluate the solution at that cell.

$$\frac{\overline{P}_{i+1,j} - 2\overline{P}_{i,j} + \overline{P}_{i-1,j}}{\triangle x^2} + \frac{\overline{P}_{i,j+1} - 2\overline{P}_{i,j} + \overline{P}_{i,j-1}}{\triangle y^2} = S_{i,j} \tag{1.1}$$

The flux integral is approximated using the control volume averages from surrounding cells. The solution is computed for each cell using the discretized equation 1.1. $S_{i,j}$ is the source term and $\overline{P}_{i,j}$ is the new solution at i,j that is calculated based on surrounding cells $\overline{P}_{i+1,j}, \overline{P}_{i-1,j}, \overline{P}_{i,j+1}$ and $\overline{P}_{i,j-1}$. The program will implement boundary conditions using ghost cells allowing the interior scheme to remain the same during calculation of bordering cells.

$$\overline{P}_{i,0} = \overline{P}_{i,1} - \triangle y g(x) \tag{1.2}$$

$$\overline{P}_{i,0} = 2\overline{P}_{i,\frac{1}{2}} - \overline{P}_{i,1} \tag{1.3}$$

The value for a ghost cell implementing Dirichlet Boundary at $y = 0$ is calculated using equation 1.2, while for a Neumann Boundary at $y = 0$ equation 1.3 is used.

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.

2.2. Laplace Problem

The program was used to solve and find solutions to the Laplace equation with mixed boundary conditions. The exact solution to this problem is written as $T(x, y)$ which will enabled a comparison for solution error assessment. A diagram shows the problem at hand in figure 2.1

$$T(x,y) = \frac{\cos(\pi x)\sinh(\pi y)}{\sinh\pi}$$

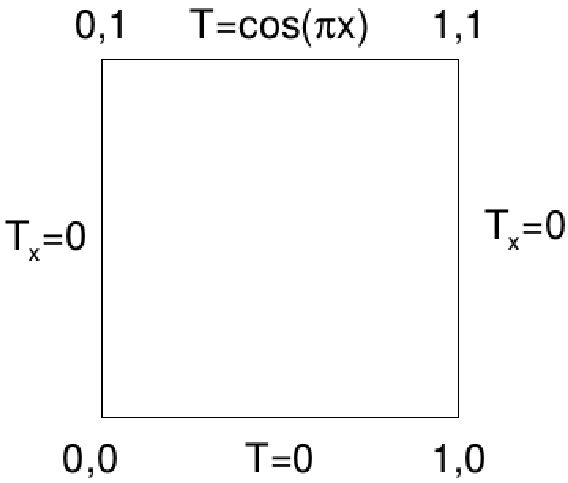


Figure 2.1: Problem uses the Laplace equation with mixed boundary conditions, The exact solution to this problem is written as $T(x,y)$

2.3. Boundary Conditions

Boundary Conditions are implemented through the calculation of the ghost cells. The ghost cell is calculated such that the boundary is implemented at the face between the ghost cell and first border cell. Below shows the function that implements boundaries for the Laplace problem.

```
for(int i = 1; i < myarray.sizex-1; ++i)
{
float d = (i-0.5)*DIM1;
//dirichlet boundaries
myarray.mcell[i][myarray.sizex-1] = 2.0*(cos(PI*d)) -
    myarray.mcell[i][myarray.sizex-2];//bottom
myarray.mcell[i][0] = -myarray.mcell[i][1];//top ghost cells
//neumann boundaries
myarray.mcell[0][i] = myarray.mcell[1][i];//left ghost cells
myarray.mcell[myarray.sizey-1][i] = myarray.mcell[myarray.sizey-2][i];//right
}
```

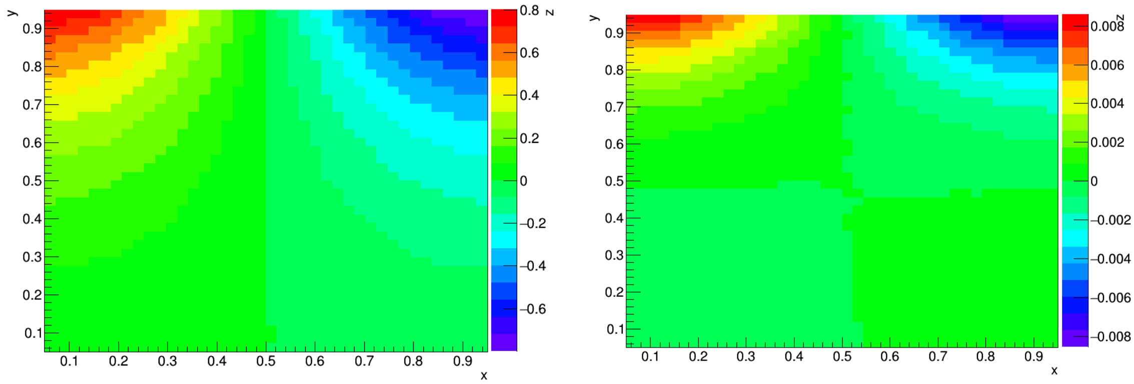
This loop is only valid for perfect square meshes. If a problem required solving a mesh with different maximum x and y cells, this loop would need to be split into two steps, one for x and one for y. Additionally this loop will need to be modified when the program is applied to different problems and at this stage is not changeable during run-time.

2.4. Iteration Scheme

A Gauss-Seidel routine was implemented to solve the system of equations approximating the solution at each grid point which is denoted as function gs_ iter_ SOR(). The routine starts from grid point 0,0 and iterates first through all j for steps of i and then through all i for steps of j. Below shows an example of a single Gauss-Seidel iteration.

```
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;
   }
}
```

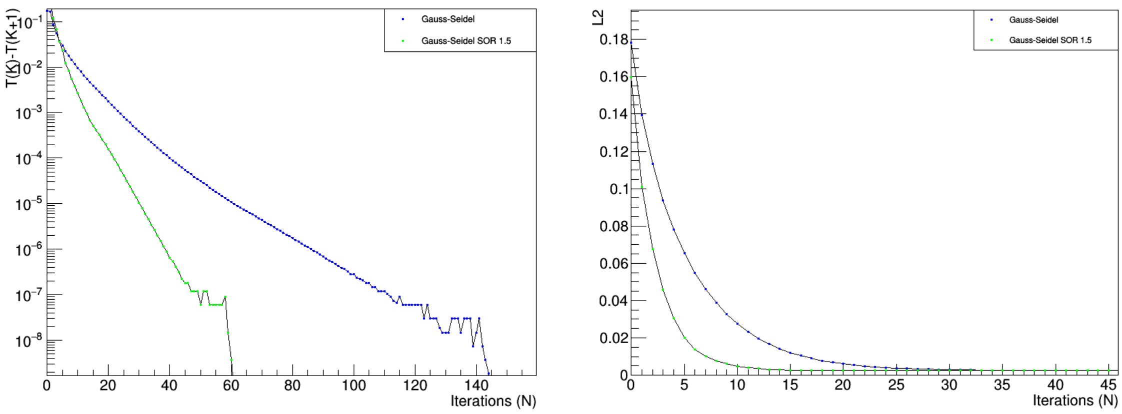
The Ghost cells are set outside of the GS-iteration in order to the improve speed of the algorithm. Setting them after each new cell calculation had little effect on the solution error or the convergence of the final solution. The function solve_ arraySOR() reiterates the Gauss-Seidel routine until the maximum solution change is below the define threshold. The gs_ iter_ SOR() routine has the ability to perform over-relaxation given parameter ω for values greater than 1. An ω value of 1 indicates over-relaxation is not being implemented. Once two GS iterations has occurred a loop determines the maximum solution change between the previous and new solution.



(a) (b)

Figure 2.2: The Numerical Solution (a) and Solution Error (b) to Laplace Problem for 10x10 mesh

The full solution was numerically calculated and is display in figure 2.2 along with the solution error (computed - exact). The maximum change in solution and the L^2 norm as a function of iteration for a 10 x 10 mesh with $\omega = 1.0, 1.5$ are displayed in figure 2.3. The exact amount of iterations to solve this problem for a 10x10 mesh was found to be 137 and 57 for $\omega = 1$ and $\omega = 1.5$ respectively when maximum solution threshold was set to 10^{-7}.



(a) (b)

Figure 2.3: Maximum solution difference (a), and the L^2 norm of the error in converged solution (b) for $\omega = 1, 1.5$ for a 10x10 mesh

The L^2 norm was found to be 2.45x10^{-3} for both cases. It clear from the plots of both the error difference and the l2 norm that the method employing over-relaxation method drastically reduces the computation time for the same problem. Its also noticed the L^2 norm also converges faster with the over-relaxation method. One would assume higher values of ω would result in faster and shorter iteration times however with ω values above 1.5 the scheme can become unstable. Essentially the change between the previous and current solution is too great and eventually the routine gets to a point were it constantly over shoots the convergence point without being able to reduce maximum solution change. In this case the ω value needs to be reduce in order to attain lower values for the solution change. The function solve_ arraySOR() contains a small subroutine which checks for conditions that might indicate a unstable condition, and in the event reduces the ω value. For future operations a function should be written the decreases the ω as a function of its tolerance, as the current method is not sufficient for predicting all situations of instability.

2.6. Convergence Behavior and Accuracy

In order to gauge the convergence behavior of the problem the program was run until steady state for ω values of 1, 1.3, and 1.5. The maximum change in solution at each iteration was plotted as a function of iteration count displayed in figure 2.4. The total iterations were found to be 472, 310, and 226 for ω values of 1, 1.3, and 1.5 respectively. This data shows the effect of over-relation (SOR) on computing time. Although the speed of the program can vastly be increased this is only true for stable vales of ω. When unstable a solution may never be reached within a define maximum change in solution, and therefore would take many more iterations than stable schemes. Generally when $\omega > 1.5$ instability should be expected for the Gauss-Seidel with over-relaxation.

Mesh Size	L^2**norm**	$\triangle x$	**Order**	**Ratio**
10 x 10	$2.457 * 10^{-3}$	0.1	-	-
20 x 20	$6.384 * 10^{-4}$	0.05	1.94	3.84
40 x 40	$1.603 * 10^{-4}$	0.025	1.96	3.98
80 x 80	$3.827 * 10^{-5}$	0.0125	2.00	4.18

Table 2.2: Table of L^2 norms for increasing mesh size

Tolerance	L^2**norm**	ω
10^{-6}	$1.767*10^{-4}$	1.0
10^{-7}	$1.605*10^{-4}$	1.0
10^{-8}	$1.603*10^{-4}$	1.0
10^{-9}	$1.603*10^{-4}$	1.0
10^{-10}	$1.603*10^{-5}$	1.0
10^{-7}	$1.603872*10^{-5}$	1.2
10^{-7}	$1.603879*10^{-5}$	1.4
10^{-7}	$1.603896*10^{-5}$	1.6

Table 2.3: Table of L^2 norms in relation to tolerance and over-relaxation parameter ω

The accuracy of the program was estimated by calculating and tabulating the L^2 norms of solution error for multiple mesh sizes. Using table 2.2 we can estimate the order of error by plotting $log(error)$ vs $log(\triangle x)$ and calculating the slope. The slope and therefore the order of accuracy was estimated to be 2.003 which rounds to 2nd order accuracy. The ratio between the errors is also indicated in table 2.2, which gives us an indication of how the error is reduced relative to mesh refinement. In this case the effect of mesh refinement can be generalized in doubling the resolution we find the L^2 error is decrease by a quarter. Additionally as we refine the mesh se find that order converges on the order estimated by the selected estimation of solution fluxes which was 2nd order. The maximum solution change was lowered to 10^{-9} in order to ensure the error being measured was discretization error and not a lack of convergence. The error trend was also examined for the algorithm variables ω and the Convergence Tolerance which is displayed in table 2.3. This table shows reducing the tolerance generally reduces the discretization error.

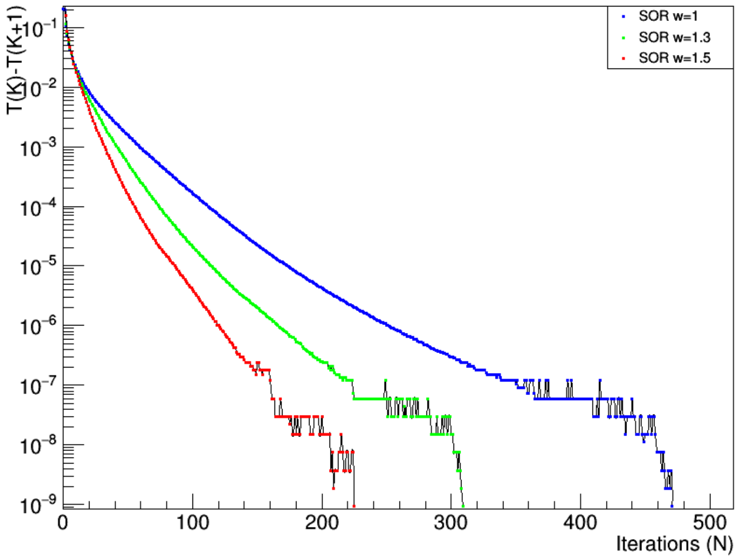


Figure 2.4: The maximum change in solution for $\omega = 1, 1.3, 1.5$ on a 20 x 20 mesh

3. Poisson Problem

3.1. Overview

Calculating pressure for in-compressible flows is done by solving the Poisson problem for pressure. Starting with the momentum equations one can obtain a simplified expression for Poisson problem (equation 3.1). This equation allows us to simply add a source term (equation 3.2) to our existing Laplace code and compute the pressure for the previous square domain as displayed in equation 1.1.

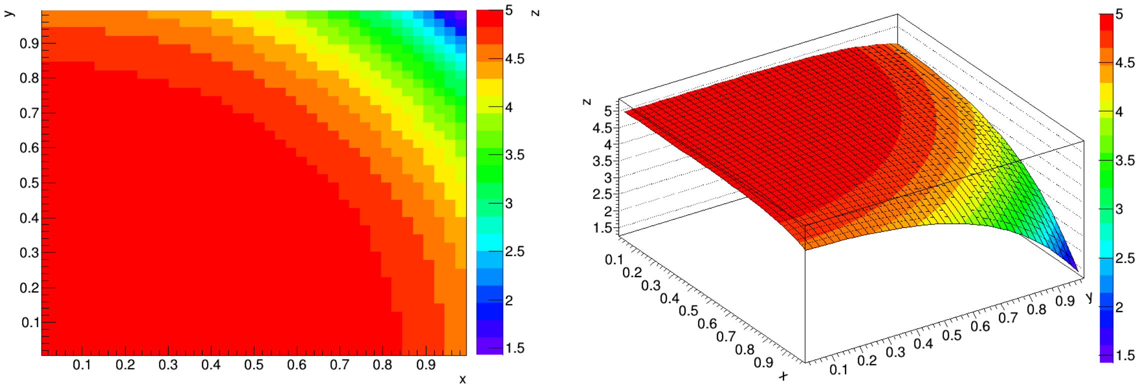
$$\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} = -\left(\left(\frac{\partial u}{\partial x}\right)^2 + 2\frac{\partial v}{\partial x}\frac{\partial u}{\partial y} + \left(\frac{\partial v}{\partial y}\right)^2\right) \tag{3.1}$$

$$S_{ij} = (3x^2 - 3y^2)^2 + 2(36x^2y^2) + (3y^2 - 3x^2) \tag{3.2}$$

For boundary conditions $\frac{\partial P}{\partial n} = 0$ at $x = 0$ and $y = 0$, and $P = 5 - \frac{1}{2}(1 + y^2)^3$ for $x = 1$ and $P = 5 - \frac{1}{2}(1 + x^2)^3$ for $y = 1$ were implemented.

3.2. Results

The Full solution to the Poisson problem was computed using $\omega = 1.3$ and is displayed in figure 3.1. This data was used to estimate the value for P at $(x, y) = (\frac{1}{2}, \frac{1}{2})$ from a 10x10 up to a 160x160 grid. The value of P was estimated using the average of the four surrounding cells. The solutions for P at $(x, y) = (\frac{1}{2}, \frac{1}{2})$ are tabulated in table 3.1 as P_1, P_2, and P_3 for decreasing grid size. The solution was found to be 4.9385 with a GCI_{fine} uncertainty of %0.0043 in the fine-grid solution. The uncertainty was calculated using the procedure described in the ASME solution accuracy handout.[Celik, 2006] The local order of accuracy for $P(\frac{1}{2}, \frac{1}{2})$ was found on average to be 1.99 which sets the accuracy at 2nd order. Table 3.2 displays the accuracy behavior with mesh refinement for the Poisson problem.



(a) (b)

Figure 3.1: Calculated Solution of Poisson problem for 20 x 20 grid

Values	$P(\frac{1}{2}, \frac{1}{2})$
N_1, N_2, N_3	40, 20, 10
r_{21}	1.909091
r_{32}	1.833333
P_1	4.937872
P_2	4.938539
P_3	4.941434
p	2.468
P_{ext}^{21}	4.937703
e_a^{21}	%2.684
e_{ext}^{21}	%0.0034
GCI_{fine}^{21}	%0.0043

Table 3.1: Table of L^2 norms for increasing mesh size

Mesh Size	Solution	Apparent Order	Rel Error	Extrap Value	GCI
10 x 10	4.941434	-	-	-	-
20 x 20	4.938539	-	0.0058%	-	-
40 x 40	4.937872	2.468819	0.0037%	4.937703	0.0043%
80 x 80	4.938059	1.895416	0.0037%	4.937605	0.0019%
160 x 160	4.938802	1.624394	0.015%	4.936539	0.0315%

Table 3.2: Table of L^2 norms for increasing mesh size

Decreasing the grid size will help the solution converge closer to the exact values. If we image the grid size becoming infinitely small the grid would become continuous and essentially would describe the exact solution. However Decreasing grid size (increasing array size) has a hard hit on computational performance. Therefore before solving such a problem one should consider the exactness required for their problem and apply the appropriate mesh size. In the case where the mesh density can vary through out the problem domain it makes sense to refine the mesh in the areas of interest while providing a course mesh for areas that are understood relatively well. This will ensure the problem is solved the fastest while attaining the best approximate solution.

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 unsolvable 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 \nGCI21 %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