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

Jerin Roberts

October 12, 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 $\overline{P}_{i,j-1}$.

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 estimated L^2 norm error between two meshs for Poisson problem

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

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

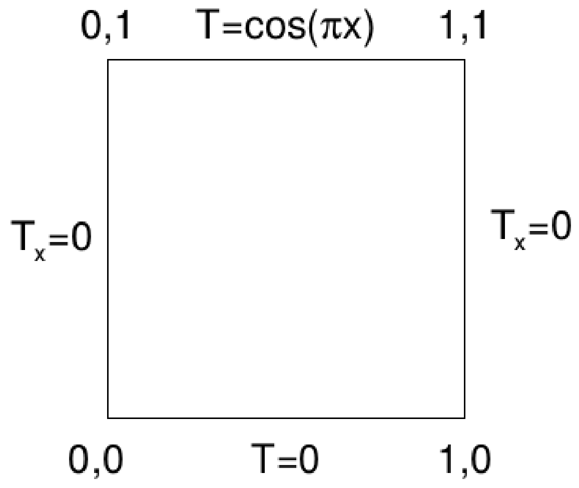
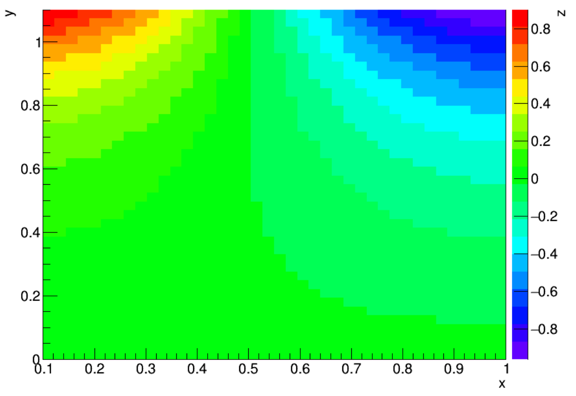


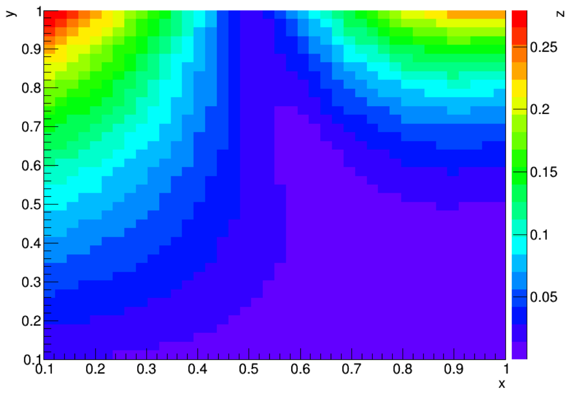
Figure 2.1: Problem uses the Laplace equation with mixed boundary conditions, The exact solution to this problem is written as $T(x, y)$

2.3. 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. Each time the routine calculates a new value it updates and reapplies the boundary conditions. No difference was noticed when boundaries conditions were only applied one per iteration, which means it could be moved outside the subroutines to reduce computation steps. Additionally the function determines the maximum solution change for each iteration. 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 as there is no amplification effect.

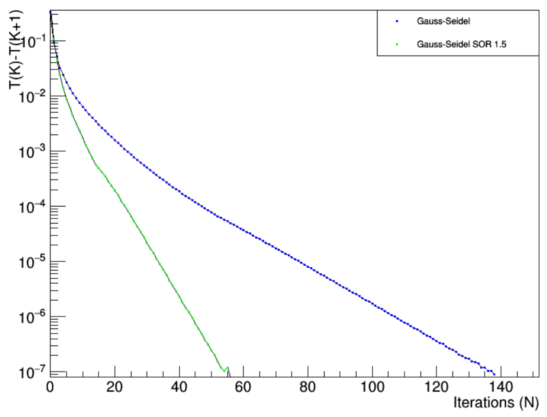


(a)

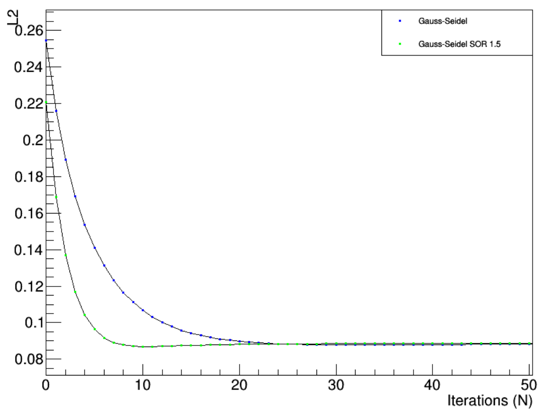


(b)

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



(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 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}. The L^2 norm was found to be 8.8557x10^{-2} 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. One would assume higher values of ω would result in faster and shorter iteration times however with ω values above 1.5 the scheme becomes 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.

2.4. 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 400, 243, and 182 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**	$\triangle x$
10 x 10	0.088557	0.1
20 x 20	0.045121	0.05
40 x 40	0.022718	0.025
80 x 80	0.011385	0.0125

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

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 0.9868 which rounds to 1st order accuracy. 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.

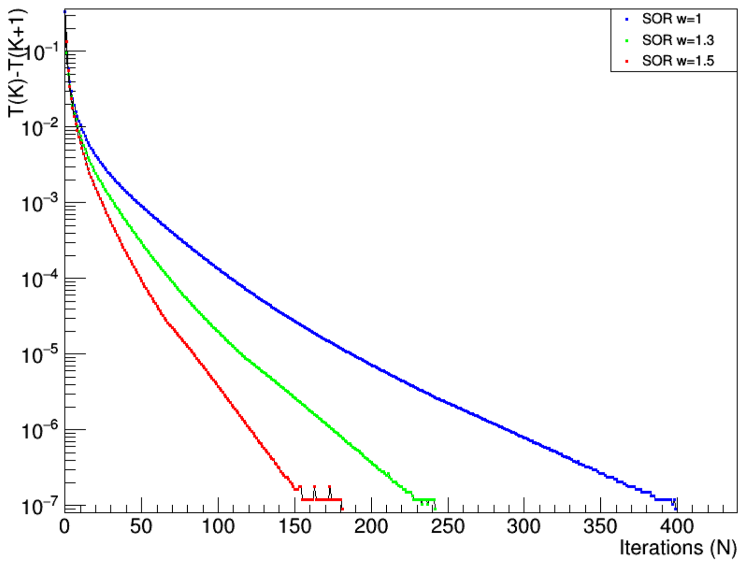


Figure 2.4: The maximum change in solution for $\omega = 1, 1.3, 1.5$ on a 20 x 20 mesh

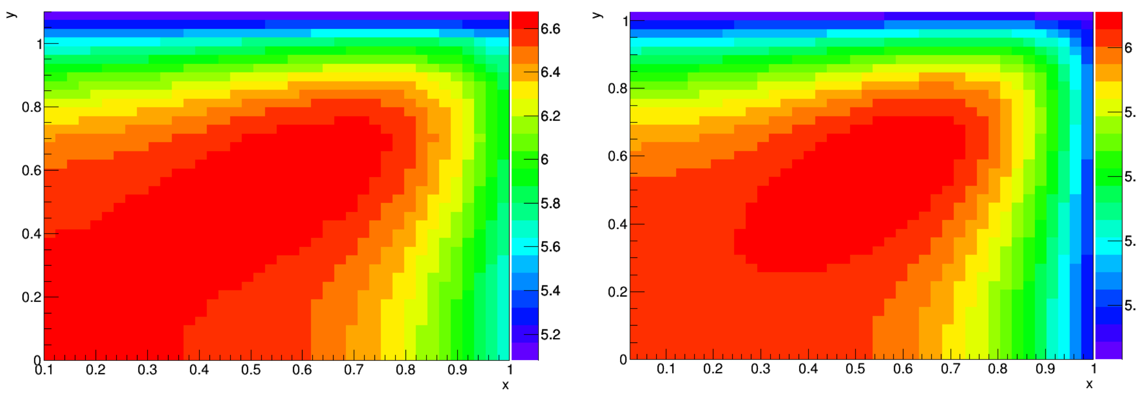
2.5. Poisson Problem

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 2.1). This equation allows us to simply add a source term (equation 2.2) to our existing Laplace code and compute the pressure for the previous square domain as displayed in equation 1.1.

$$\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} = -\left(\left(\frac{\partial u}{\partial x}\right)^2 + 2\frac{\partial v}{\partial x}\frac{\partial u}{\partial y} + \left(\frac{\partial v}{\partial y}\right)^2\right) \tag{2.1}$$

$$S_{ij} = (3x^2 - 3y^2)^2 + 2(36x^2y^2) + (3y^2 - 3x^2) \tag{2.2}$$

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. The Full solution to the Poisson problem was computed using $\omega = 1.3$ and is displayed in figure 2.5. This data was used to estimate the value for P at $(x, y) = (\frac{1}{2}, \frac{1}{2})$ for a 10x10, 20x20 and 40x40 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 2.3 as P_1, P_2, and P_3 for decreasing grid size. The solution was found to be 6.098 with a GCI_{fine} uncertainty of %2.24 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 to be 2.829 which sets the accuracy of the value between 2nd and 3rd order. 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. This will ensure the problem is solved the fastest while attaining the best approximate solution.



(a) (b)

Figure 2.5: Calculated Solution for 20 x 20 grid (a) and 40 x 40 grid (b)

Values	$P(\frac{1}{2},\frac{1}{2})$
N_1, N_2, N_3	40, 20, 10
r_{21}	1.381
r_{32}	1.354
P_1	6.098
P_2	6.262
P_3	6.633
p	2.829
P^{21}_{ext}	5.989
e^{21}_{a}	%2.684
e^{21}_{ext}	%1.826
GCI^{21}_{fine}	%2.242
$L^2_{21} norm$	0.3528
$L^2_{32} norm$	0.3082

Table 2.3: Table of L^2 norms for increasing mesh size

3. 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
```

```
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, 10, 10, 1.0);//array, xsize, ysize, dimension
set_array_size(poisson2, 40, 40, 1.0);
set_array_size(poisson3, 60, 60, 1.0);


//set ghost cells as boundary conditions


set_zero(poisson1);//zero
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, E07, 1.3);
cout << "Solution: " << get_solution(poisson1) << "\n";


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

solve_arraySOR(poisson2, E07, 1.3);
cout << "Solution: " << get_solution(poisson2) << "\n";

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

solve_arraySOR(poisson3, E07, 1.3);
cout << "Solution: " << get_solution(poisson3) << "\n";

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

get_discrete_Error(poisson1, poisson2, poisson3, 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 162
#define maxy 162
#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);

void set_ghostcells(carray & myarray);

void set_zero(carray & myarray);

void print_array(carray & myarray);

float gs_iter_SOR(carray & myarray, float omega);

float calc_source(carray & myarray, int i, int j);

float calc_newcell(carray & myarray, float source, float Tip1_j, float Tim1_j,
    float Ti_jp1 ,float Ti_jm1);

void get_surcells(carray & myarray, float & Tip1_j, float & Tim1_j, float &
    Ti_jp1 ,float & Ti_jm1, int i, int j);

void solve_arraySOR(carray & myarray, float E0, float w);

void get_discrete_Error(carray ray1, carray ray2, carray ray3, float DIM);

float get_solution(carray & myarray);


#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*DIM1;
   myarray.mcell[i][0] = myarray.mcell[i][1];//top boundary
   myarray.mcell[0][i] = myarray.mcell[1][i];//left boundary
   myarray.mcell[i][myarray.sizex-1] = 5+((1/2)*pow((1+pow(d, 2)),3));//bottom
       boundary
   myarray.mcell[myarray.sizey-1][i] = 5+((1/2)*pow((1+pow(d, 2)),3));//right
       boundary
        }
}



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

```
float gs_iter_SOR(carray & myarray, float omega)
{
float DIM1 = myarray.DIM1;//get dimensions of array (not grid size)
float maxdiff = -1;
float Tip1_j, Tim1_j, Ti_jp1, Ti_jm1;//define values for surrounding cells
float l2sum = 0;
float sx = myarray.sizex-2;
float sy = myarray.sizey-2;

float dx = 0;//change in x
float dy = 0;//change in y



//-----iterate through all x for steps y -----//

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


     for(int i = 1; i < myarray.sizex-1; ++i)
     {
     dx = DIM1*i;
     dy = DIM1*j;

     //----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);

     //-----find difference between new and old cell-----//
     float diff = abs(newcell - myarray.mcell[i][j]);
     if(diff > maxdiff)
     maxdiff = diff;


     //-----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 = DIM1*i;
     dy = DIM1*j;

     //----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);

     //-----find difference between new and old cell-----//
     float diff = abs(newcell - myarray.mcell[i][j]);
     if(diff > maxdiff)
     maxdiff = diff;


     //-----update current cell----//
     myarray.mcell[i][j] = newcellSOR;
     set_ghostcells(myarray);
     }

   }

myarray.diff.push_back(maxdiff);
++myarray.iterations;

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

void 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";
}

//---------------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";
       update2 = update2 + 100;
      }

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

      if(div > 20 && relax_on && w > 1.3)//reduces over-relaxation for high w
          when unstable
      {
```

```
      w = 1.3;
      relax_on = false;
      cout << "Relaxation Reduced to 1.0 @ " << 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 = DIM1*i;
float dy = DIM1*j;
float source =
    -1*(pow(3*pow(dx,2)-3*pow(dy,2),2)+72*(pow(dx,2)*pow(dy,2))+pow(3*pow(dy,2)-3*pow(dx,2),2));
//source = 0 for Laplace problem

return source;
}

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

float get_solution(carray & myarray)
{
int sx = (myarray.sizex-2)/2;
int sy = (myarray.sizey-2)/2;
float sol =
    (myarray.mcell[sx-1][sy+1]+myarray.mcell[sx+1][sy+1]+myarray.mcell[sx+1][sy-1]+myarray.mcell[

//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 = sqrt((1.0/ray1.sizex)*(DIM));
float h2 = sqrt((1.0/ray2.sizex)*(DIM));
float h3 = sqrt((1.0/ray3.sizex)*(DIM));

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


printf("h1: %f h2: %f h3: %f, sol1: %f sol2: %f sol3: %f\n",h1, h2, h3, sol1,
    sol2, sol3);

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

printf("r32: %f r21: %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("phi_ext21: %f phi_ext32 %f\n", sol_ext21, sol_ext32);

float ea21 = abs((sol1-sol2)/sol1);
float ea32 = abs((sol2-sol3)/sol2);

float e_ext21 = abs((sol_ext21-sol1)/sol_ext21);
float e_ext32 = abs((sol_ext32-sol2)/sol_ext32);

float GCI_21 = (1.25*ea21)/(pow(r21,p)-1.0);
float GCI_32 = (1.25*ea32)/(pow(r32,p)-1.0);

printf("ea21/32: %f %f e_ext21/32: %f %f GC121/32 %f %f\n", ea21, ea32,
    e_ext21, e_ext32, GCI_21, GCI_32);

}
```

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