Computer Problem #1

Paul Dorman

February 12, 2008

1 Introduction

In computational fluid dynamics, a fluid flow is solved numerically over a body of interest. This is done by creating a grid in the computational domain, and solving various equations at points on the grid. This problem involved creating a grid given a geometry. First the grid is created algebraically, with points on the boundary of the geometry defined, and then interpolating between them in some fashion. Once this initial grid is created, it can be refined using numerical partial differential equation methods to move the grid points around to achieve a more optimal distribution.

For this case, the Poisson Equations (show below) were used as the PDE to refine the grid:

$$\xi_{xx} + \xi_{yy} = P(\xi, \eta)$$

$$\eta_{xx} + \eta_{yy} = Q(\xi, \eta)$$

After some manipulation and discretization, these governing equations can be written in the form shown below:

$$b_{i,j}x_{i-1,j}^{n+1} + d_{i,j}x_{i,j}^{n+1} + a_{i,j}x_{i+1,j}^{n+1} = c_{i,j}$$

$$\beta_{i,j}y_{i-1,j}^{n+1} + \delta_{i,j}y_{i,j}^{n+1} + \alpha_{i,j}y_{i+1,j}^{n+1} = \gamma_{i,j}$$

The coefficients are given by the expressions below, assuming $\Delta\xi = \Delta\eta = 1$ everywhere:

$$b_{i,j} = A_1\left(1 - \frac{\phi}{2}\right) = \left(x_\eta^2 + y_\eta^2\right)\left(1 - \frac{\phi}{2}\right) = \left[\left(\frac{x_{i,j+1} - x_{i,j-1}}{2}\right)^2 + \left(\frac{y_{i,j+1} - y_{i,j-1}}{2}\right)^2\right]\left(1 - \frac{\phi}{2}\right)$$

$$d_{i,j} = -2(A_1 + A_3) = -2\left\{\left[\left(\frac{x_{i,j+1} - x_{i,j-1}}{2}\right)^2 + \left(\frac{y_{i,j+1} - y_{i,j-1}}{2}\right)^2\right] + \left[\left(\frac{x_{i+1,j} - x_{i-1,j}}{2}\right)^2 + \left(\frac{y_{i+1,j} - y_{i-1,j}}{2}\right)^2\right]\right\}$$

$$a_{i,j} = A_1\left(1 + \frac{\phi}{2}\right) = \left(x_\eta^2 + y_\eta^2\right)\left(1 + \frac{\phi}{2}\right) = \left[\left(\frac{x_{i,j+1} - x_{i,j-1}}{2}\right)^2 + \left(\frac{y_{i,j+1} - y_{i,j-1}}{2}\right)^2\right]\left(1 + \frac{\phi}{2}\right)$$

$$c_{i,j} = \frac{A_2}{2}\left(x_{i+1,j+1} - x_{i+1,j-1} - x_{i-1,j+1} + x_{i-1,j-1}\right) - A_3\left(x_{i,j+1} + x_{i,j-1}\right) - A_3\frac{\psi}{2}\left(x_{i,j+1} - x_{i,j-1}\right)$$

$$= \frac{x_\xi y_\eta + y_\xi x_\eta}{2}\left(x_{i+1,j+1} - x_{i+1,j-1} - x_{i-1,j+1} + x_{i-1,j-1}\right) - \left(x_\xi^2 + y_\xi^2\right)\left(x_{i,j+1} + x_{i,j-1}\right) - \left(x_\xi^2 + y_\xi^2\right)\frac{\psi}{2}\left(x_{i,j+1} - x_{i,j-1}\right)$$

$$= \frac{\left(\frac{x_{i+1,j} - x_{i-1,j}}{2}\right)\left(\frac{y_{i,j+1} - y_{i,j-1}}{2}\right) + \left(\frac{y_{i+1,j} - y_{i-1,j}}{2}\right)\left(\frac{x_{i,j+1} - x_{i,j-1}}{2}\right)}{2}\left(x_{i+1,j+1} - x_{i+1,j-1} - x_{i-1,j+1} + x_{i-1,j-1}\right)$$

$$-\left[\left(\frac{x_{i+1,j}-x_{i-1,j}}{2}\right)^2+\left(\frac{y_{i+1,j}-y_{i-1,j}}{2}\right)^2\right](x_{i,j+1}+x_{i,j-1})-\left[\left(\frac{x_{i+1,j}-x_{i-1,j}}{2}\right)^2+\left(\frac{y_{i+1,j}-y_{i-1,j}}{2}\right)^2\right]\frac{\psi}{2}(x_{i,j+1}-x_{i,j-1})$$

The β, δ, and α coefficients are the same as the b, d, and a coefficients:

$$\beta_{i,j}=A_1\left(1-\frac{\phi}{2}\right)=\left(x_\eta^2+y_\eta^2\right)\left(1-\frac{\phi}{2}\right)=\left[\left(\frac{x_{i,j+1}-x_{i,j-1}}{2}\right)^2+\left(\frac{y_{i,j+1}-y_{i,j-1}}{2}\right)^2\right]\left(1-\frac{\phi}{2}\right)$$

$$\delta_{i,j}=-2\left(A_1+A_3\right)=-2\left\{\left[\left(\frac{x_{i,j+1}-x_{i,j-1}}{2}\right)^2+\left(\frac{y_{i,j+1}-y_{i,j-1}}{2}\right)^2\right]+\left[\left(\frac{x_{i+1,j}-x_{i-1,j}}{2}\right)^2+\left(\frac{y_{i+1,j}-y_{i-1,j}}{2}\right)^2\right]\right\}$$

$$\alpha_{i,j}=A_1\left(1+\frac{\phi}{2}\right)=\left(x_\eta^2+y_\eta^2\right)\left(1+\frac{\phi}{2}\right)=\left[\left(\frac{x_{i,j+1}-x_{i,j-1}}{2}\right)^2+\left(\frac{y_{i,j+1}-y_{i,j-1}}{2}\right)^2\right]\left(1+\frac{\phi}{2}\right)$$

The γ coefficient is slightly different, however, using y instead of x:

$$\gamma_{i,j}=\frac{A_2}{2}\left(y_{i+1,j+1}-y_{i+1,j-1}-y_{i-1,j+1}+y_{i-1,j-1}\right)-A_3\left(y_{i,j+1}+y_{i,j-1}\right)-A_3\frac{\psi}{2}\left(y_{i,j+1}-y_{i,j-1}\right)$$

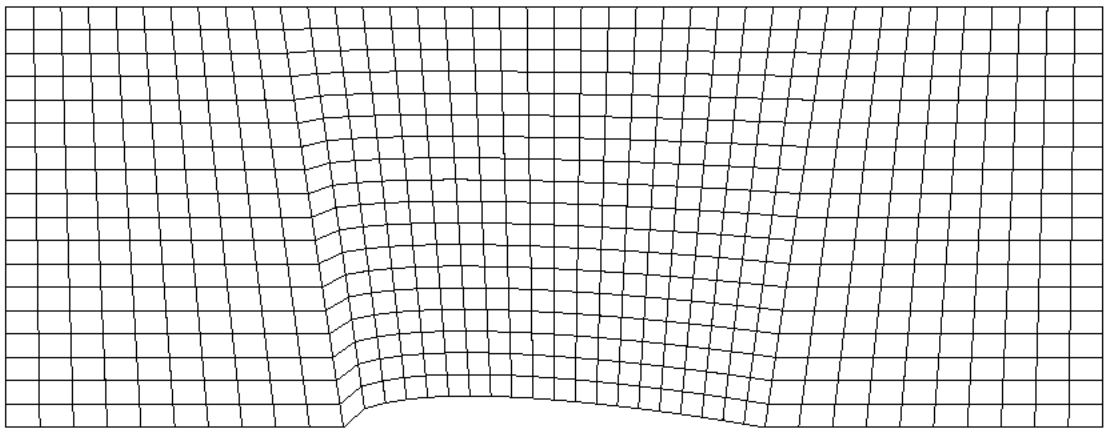
$$=\frac{x_\xi y_\eta+y_\xi x_\eta}{2}\left(y_{i+1,j+1}-y_{i+1,j-1}-y_{i-1,j+1}+y_{i-1,j-1}\right)-\left(x_\xi^2+y_\xi^2\right)\left(y_{i,j+1}+y_{i,j-1}\right)-\left(x_\xi^2+y_\xi^2\right)\frac{\psi}{2}\left(y_{i,j+1}-y_{i,j-1}\right)$$

$$=\frac{\left(\frac{x_{i+1,j}-x_{i-1,j}}{2}\right)\left(\frac{y_{i,j+1}-y_{i,j-1}}{2}\right)+\left(\frac{y_{i+1,j}-y_{i-1,j}}{2}\right)\left(\frac{x_{i,j+1}-x_{i,j-1}}{2}\right)}{2}\left(y_{i+1,j+1}-y_{i+1,j-1}-y_{i-1,j+1}+y_{i-1,j-1}\right)$$

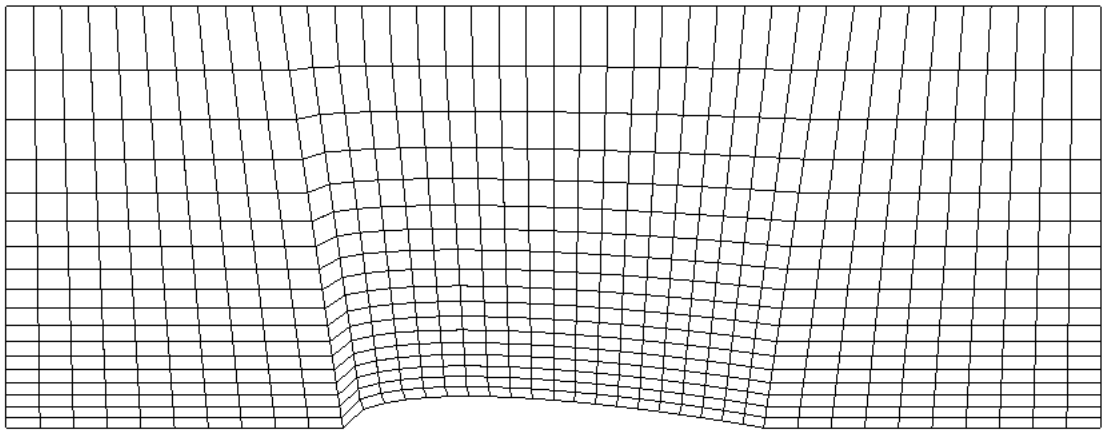
$$-\left[\left(\frac{x_{i+1,j}-x_{i-1,j}}{2}\right)^2+\left(\frac{y_{i+1,j}-y_{i-1,j}}{2}\right)^2\right](y_{i,j+1}+y_{i,j-1})-\left[\left(\frac{x_{i+1,j}-x_{i-1,j}}{2}\right)^2+\left(\frac{y_{i+1,j}-y_{i-1,j}}{2}\right)^2\right]\frac{\psi}{2}(y_{i,j+1}-y_{i,j-1})$$

2 Grids

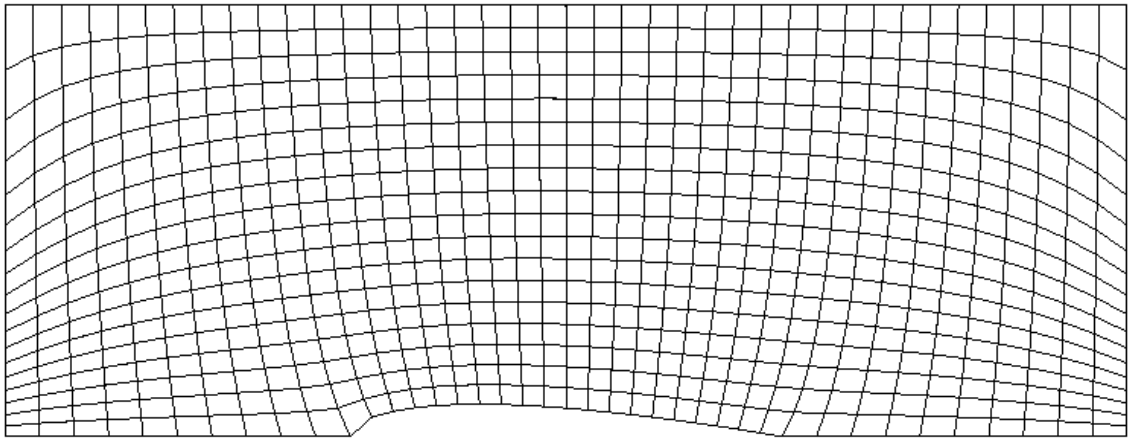
The first grid, shown below, is an initial algebraic grid, with spacing factor $c_y = 0.001$:



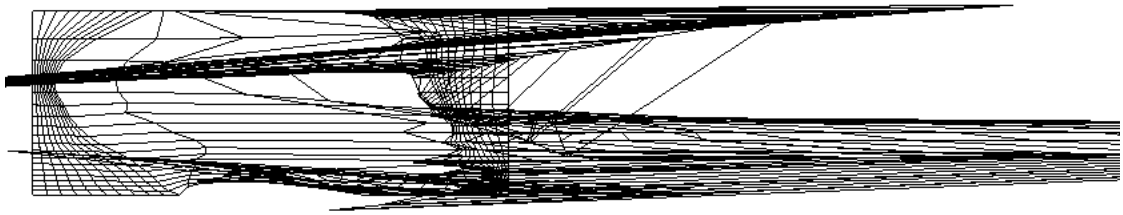
The second grid, show below, is an initial algebraic grid, with spacing factor $c_y = 2.0$:



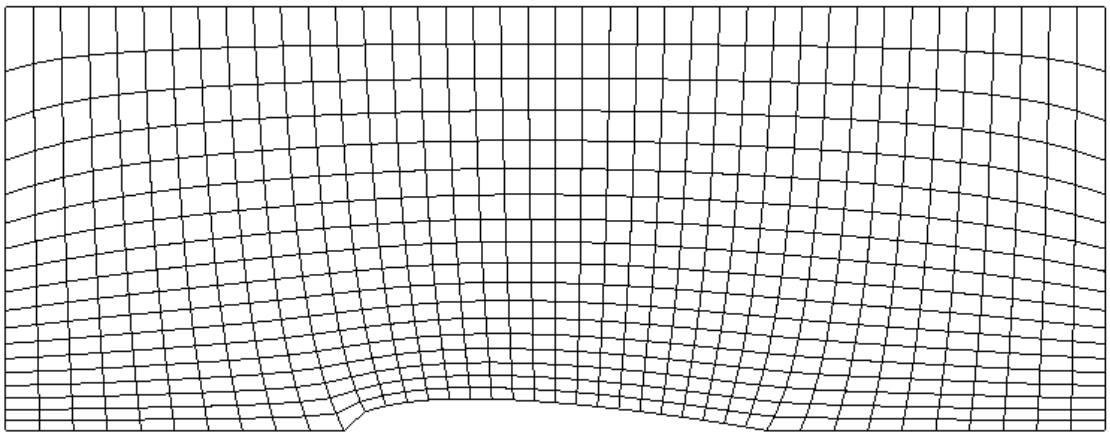
The third grid, show below, began with the second algebraic grid (above), and then applied the PDE refinement using $\phi = 0$ and $\psi = 0$:



The fourth grid also began with the second algebraic grid, but used the ϕ and ψ values given in the assignment. Because of the appearance of the resulting grid, I believe I must have misimplemented something in the code. Parts of the initial grid can still be seen on the left-hand side, but many of the points are distributed about at tremendous distances outside the boundaries:

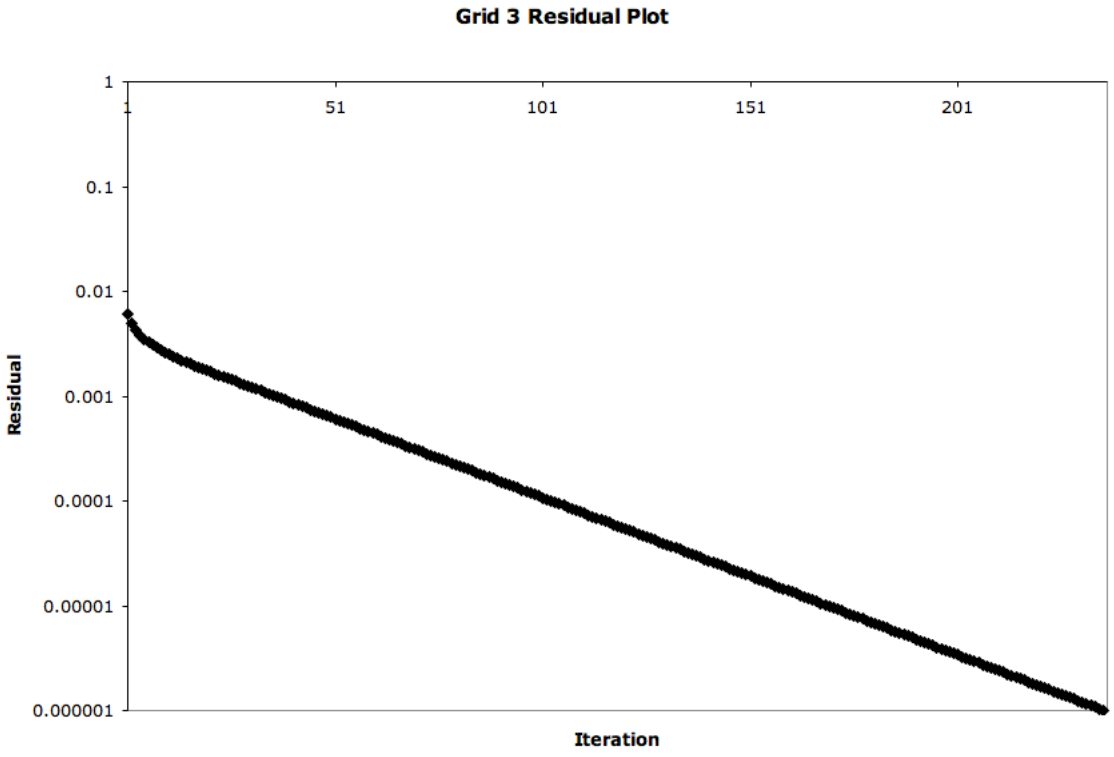


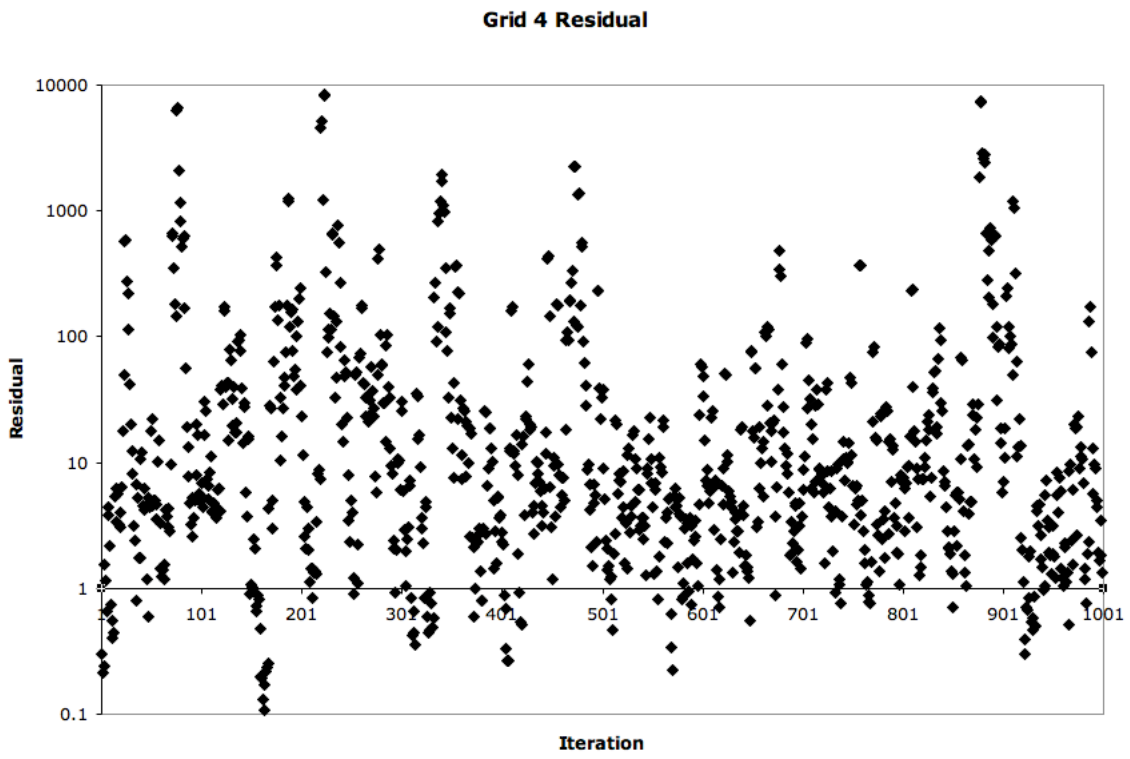
The fifth and final grid is shown below:

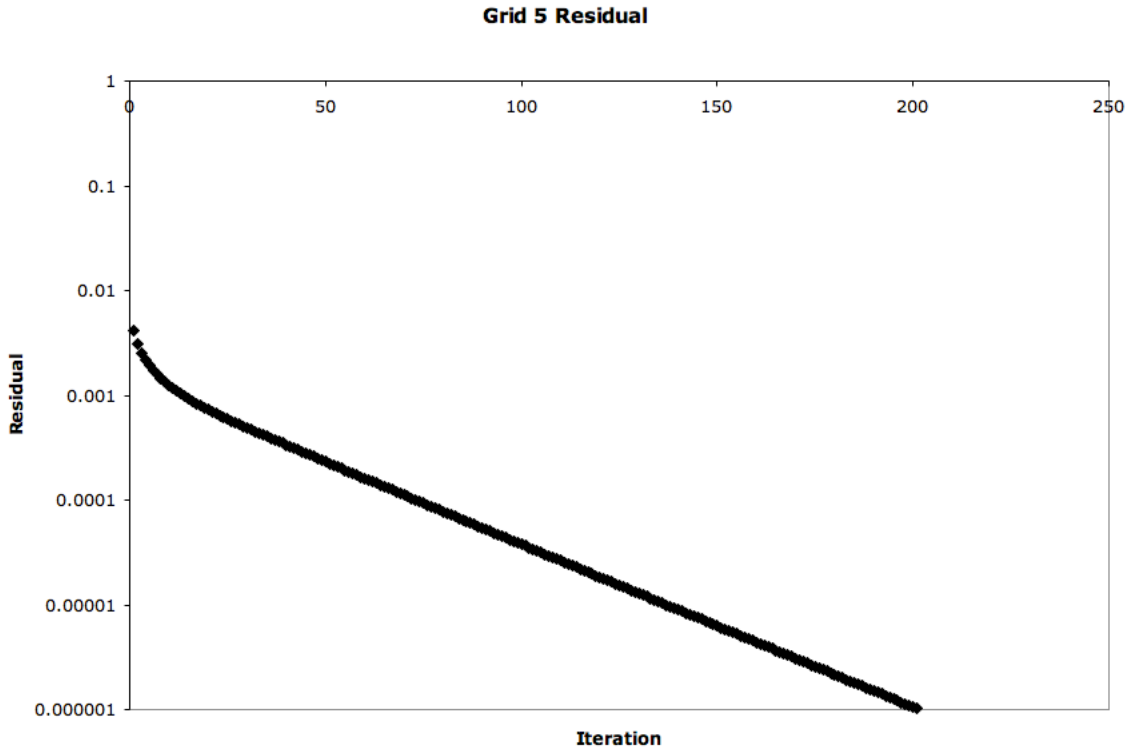


3 Convergence

For each of the PDE grids (grids 3-5), the convergence criterion was that the root mean square residual be less than or equal to 1E-6. Plots for each are shown below. As can be seen, the residuals for grids 3 and 5 converged logarithmically. However, the residual for grid 4, which was the problematical grid shown above, displays no signs of convergence after 1000 iterations.




Grid 4 Residual
10000
1000
100
10
1
0.1
Residual
101
201
301
401
501
601
701
801
901
1001
Iteration


Grid 5 Residual
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
Residual
0
50
100
150
200
250
Iteration

4 Results

The algebraic grids represent the roughest, but simplest attempts at distributing the grid points. The first one, with no clustering, represents a constant, linear interpolation between grid points. This does not allow the advantages of clustering which are desired where flow gradients are steeper.

Grid 2 uses clustering, distributing more points toward the bottom of the solution space, where the gradients will be highest. Unfortunately, it also introduces much skewness into the cells, especially where it transitions between the horizontal beginning and end and the middle airfoil section.

Grid 3, which had the control terms set to 0, became the solution to the Laplace Equation. Although it started out with the algebraic distribution still visible on the boundaries, the tendency was to evenly distribute the grid points as much as possible given the constraints set by the boundary points, giving results similar to those of the first algebraic grid, although with somewhat less skewness.

Grid 4 obviously has severe problems. Because grids 3 and 5 turned out well, I am forced to believe I have correctly implemented the Thomas algorithm, leaving the likelihood that I somehow erred in writing the control terms.

Grid 5 was supposed to be the best possible grid for inviscid, subsonic flow. As always with CFD, it is desirable to cluster the grid points where the flow gradients are steepest. In this case, that is near the airfoil. Since this is inviscid, subsonic flow there is neither a boundary layer nor any shocks. However, the flow is still experiencing the strongest gradient nearest to the airfoil. The algebraic grid given using the spacing $c_y = 2.0$ gives a good start at clustering grid points with this in mind. Unfortunately, the Laplace grid solution, seen in Grid 3, undoes the clustering effect, distributing the grid points more evenly than is desirable. Consequently, for this grid I attempted to find control terms that would reinstitute the clustering found in the initial algebraic grid, with grid lines of constant η clustered more tightly toward the bottom of the grid, over the airfoil surface. In trying various combinations, I found that the ϕ control term tended to control the horizontal distribution of grid points. Seeing no real advantage in clustering in this direction, I left this function at 0 as in the Laplace grid. But for the ψ term, which I found tended to control the vertical distribution, I found that setting it equal to $-x_\xi$ produced the desired effect, clustering grid points toward the bottom, and gradually spacing them out toward the top. Since I began with the same clustered algebraic grid for both Grid 3 and Grid 5, and the clustering is maintained on the interior in Grid 5, this also helps to reduce skewness relative to Grid 3.

5 Using the Code

I created this code on a computer running Mac OS X 10.4, using the GCC 4.0.1 compiler. All the variables that the user might change can be found toward the top of the code, immediately above the *main* function. These variables are *cy*, specifying the algebraic clustering factor; *maxItr*, giving the maximum number of iterations that should be run if convergence is not achieved; *tol,* which specifies the maximum value at which the RMS residual will be considered converged; and *custom*, which designates which PDE should be run. This last variable, *custom*, should be set to *0* to run the Laplace case, *1* to run the unsuccessful case given in this assignment, or *2* to run the case I found to be best.

The code will create three files: *Residual.csv*, an Excel file plotting the residual against the iteration number; *GRID2D-0.dat*, the initial algebraic grid, formatted for the grid visualization software provided on T-Square; and *GRID2D-*N*.dat,* the final grid, where N is either the iteration number at which convergence was achieved, or one greater than the maximum number of iterations allowed. In all cases, the maximum iteration number of 1000 as given in the variable *maxItr* is sufficient.

5.1 Grid 1

To create Grid 1, set the variable *cy* to 0.0. The output file *GRID2D-0.dat* will contain the algebraic grid for this case.

5.2 Grid 2

To create Grid 2, set the variable *cy* to 2.0. The output file *GRID2D-0.dat* will contain the algebraic grid for this case.

5.3 Grid 3

To create Grid 3, set the variable *cy* to 2.0, set *tol* to 1E-6, and set *custom* to 0. The final grid file will be created, named *GRID2D-*N*.dat*, where N is not 0, and the resulting residual will be given in the Excel file *Residual.csv.*

5.4 Grid 4

To create Grid 4, set the variable *cy* to 2.0, set *tol* to 1E-6, and set *custom* to 1. The final grid file will be created, named *GRID2D*-N.*dat*, where *N* is not 0, and the resulting residual will be given in the Excel file *Residual.csv*.

5.5 Grid 5

To create Grid 5, set the variable *cy* to 2.0, set *tol* to 1E-6, and set *custom* to 2. The final grid file will be created, named *GRID2D*-N.*dat*, where *N* is not 0, and the resulting residual will be given in the Excel file *Residual.csv*.

6 Code

```
#include <stdio.h>
#include <math.h>

typedef struct {
        double x, y;
} Point;

#define IMAX 41
#define JMAX 19
#define BUFF 10

// main

void setUpGridBoundaries();
double NACA00( double t, double x );
void fillAlgebraicGrid();
void dumpGrid( int i );
void step( int j );
double iteration();

// metrics

double dxdxi( int i, int j );
double dxdeta( int i, int j );
double dydxi( int i, int j );
double dydeta( int i, int j );
double d2xdxi2( int i, int j );
double d2xdeta2( int i, int j );
double d2xdxideta( int i, int j );
double d2ydxi2( int i, int j );
double d2ydeta2( int i, int j );
double d2ydxideta( int i, int j );

// controls

double phi( int i, int j );
double psi( int i, int j );
double A1( int i, int j );
double A2( int i, int j );
double A3( int i, int j );
double psiBoundary( int i, int j );
double phiBoundary( int i, int j );
```

```
//-------------------------------------------------------//

Point grid[IMAX][JMAX];
double residual;
double cy = 2.0;
int maxItr = 1000;
double tol = 1E-6;
int custom = 0; // 0 for Laplace, 1 for CP #1 Poisson, 2 for custom Poisson

//-------------------------------------------------------//

int main (int argc, const char * argv[]) {
        int i;
        double r;
        FILE *rPlot = fopen( "Residual.csv", "w" );

        setUpGridBoundaries();
        fillAlgebraicGrid();

        dumpGrid( 0 );
        fprintf( rPlot, "Iteration,Residual\n" );

        for( i = 1; i <= maxItr; i++ ){
                if( (r = iteration()) <= tol )
                        break;
                fprintf( rPlot, "%d,%g\n", i, r );
        }

        dumpGrid( i );

        printf( "Final Residual: %f\n", r );

    return 0;
}

// distributes points on grid boundary
void setUpGridBoundaries(){
        int i;

        for( i = 0; i <= BUFF; i++ ){
                grid[i][0].x = -0.8 + 0.8/BUFF*i;
                grid[i][0].y = 0.0;
        }

        for( i = BUFF+1; i < IMAX-BUFF; i++ ){
                grid[i][0].x = (i-BUFF)*1.0/(IMAX-2*BUFF-1);
                grid[i][0].y = NACA00( 0.15, grid[i][0].x );
        }

        for( i = IMAX-BUFF; i < IMAX; i++ ){
                grid[i][0].x = 1.0+0.8/BUFF*(i-(IMAX-BUFF-1));
                grid[i][0].y = 0.0;
        }
```

```
        for( i = 0; i < IMAX; i++ ){
                grid[i][JMAX-1].x = -0.8 + 2.6*i/(IMAX-1);
                grid[i][JMAX-1].y = 1.0;
        }
}

// returns t(x) for a NACA 00xx series airfoil
double NACA00( double t, double x ){
        double xint = 1.008930411365;

 return 5*t*(0.2969*sqrt(xint*x)-0.126*xint*x-0.3516*pow(xint*x,2) +
0.2843*pow(xint*x,3)-0.1015*pow(xint*x,4));
}

// fills in the grid algebraically, once the boundaries are set up
void fillAlgebraicGrid(){
        int i, j;

        for( i = 0; i < IMAX; i++ ){
                for( j = 1; j < JMAX-1; j++ ){
                        grid[i][j].y = grid[i][0].y -
(grid[i][JMAX-1].y-grid[i][0].y)/cy*log(1+(exp(-cy)-1)*j/(JMAX-1));
                        grid[i][j].x = grid[i][0].x +
(grid[i][0].x-grid[i][JMAX-1].x)*grid[i][j].y/(grid[i][0].y-grid[i][JMAX-1].y);
                }
        }
}

// creates a grid file for visualization
void dumpGrid( int n ){
        FILE *file;
        int i, j;
        char fileName[32];

        sprintf( fileName, "GRID2D-%d.dat", n );

        file = fopen( fileName, "w" );

        fprintf( file, "%d  %d\n", IMAX, JMAX );

        for( j = 0; j < JMAX; j++ ){
                for( i = 0; i < IMAX; i++ ){
                        fprintf( file, "%f\n", grid[i][j].x );
                }
        }
        for( j = 0; j < JMAX; j++ ){
                for( i = 0; i < IMAX; i++ ){
                        fprintf( file, "%f\n", grid[i][j].y );
                }
        }
}

//-------------------------------------------------//
```

```
// performs the Thomas algorithm at row j
void step( int j ){
        double a[IMAX], b[IMAX], c[IMAX], d[IMAX]
        double alpha[IMAX], beta[IMAX], gamma[IMAX], delta[IMAX];
        int i;

        // set matrix boundary conditions
        a[0] = a[IMAX-1] = b[0] = b[IMAX-1] = 0;
        alpha[0] = alpha[IMAX-1] = beta[0] = beta[IMAX-1] = 0.0;
        d[0] = d[IMAX-1] = delta[0] = delta[IMAX-1] = 1.0;
        c[0] = grid[0][j].x;
        c[IMAX-1] = grid[IMAX-1][j].x;
        gamma[0] = grid[0][j].y;
        gamma[IMAX-1] = grid[IMAX-1][j].y;

        // fill matrix coefficients
        for( i = 1; i < IMAX-1; i++ ){
                b[i] = A1(i,j)*(1.0-phi(i,j)/2.0);
                d[i] = -2.0*(A1(i,j)+A3(i,j));
                a[i] = A1(i,j)*(1.0+phi(i,j)/2.0);
                c[i] = A2(i,j)/2.0*
(grid[i+1][j+1].x-grid[i+1][j-1].x-grid[i-1][j+1].x+grid[i-1][j-1].x) -
A3(i,j)*(grid[i][j+1].x+grid[i][j-1].x) -
A3(i,j)*psi(i,j)/2.0*(grid[i][j+1].x-grid[i][j-1].x);

                beta[i] = b[i];
                delta[i] = d[i];
                alpha[i] = a[i];
                gamma[i] = A2(i,j)/2.0*
(grid[i+1][j+1].y-grid[i+1][j-1].y-grid[i-1][j+1].y+grid[i-1][j-1].y) -
A3(i,j)*(grid[i][j+1].y+grid[i][j-1].y) -
A3(i,j)*psi(i,j)/2.0*(grid[i][j+1].y-grid[i][j-1].y);
        }

        // Thomas algorithm
        for( i = 1; i < IMAX; i++ ){
                d[i] = d[i] - b[i]/d[i-1]*a[i-1];
                delta[i] = delta[i] - beta[i]/delta[i-1]*alpha[i-1];

                c[i] = c[i] - b[i]/d[i-1]*c[i-1];
                gamma[i] = gamma[i] - beta[i]/delta[i-1]*gamma[i-1];
        }

        for( i = IMAX-2; i > 0; i-- ){
                double newX = (c[i] - a[i]*grid[i+1][j].x)/d[i];
                double newY = (gamma[i] - alpha[i]*grid[i+1][j].y)/delta[i];

                residual += pow( newX - grid[i][j].x, 2 ) + pow( newY - grid[i][j].y, 2 );

                grid[i][j].x = newX;
                grid[i][j].y = newY;
        }
}
```

```
// performs the Thomas algorithm over the entire grid
double iteration(){
        int j;

        residual = 0.0;

        for( j = 1; j < JMAX-1; j++ ){
                step(j);
        }

        return sqrt( residual/(2.0*(IMAX-2)*(JMAX-2)) );
}

//-----------------------------------------------//

// metrics
double dxdxi( int i, int j ){
        return (grid[i+1][j].x-grid[i-1][j].x)/2;
}

double dydxi( int i, int j ){
        return (grid[i+1][j].y-grid[i-1][j].y)/2;
}

double dxdeta( int i, int j ){
        return (grid[i][j+1].x-grid[i][j-1].x)/2;
}

double dydeta( int i, int j ){
        return (grid[i][j+1].y-grid[i][j-1].y)/2;
}

double d2xdxi2( int i, int j ){
        return grid[i+1][j].x-2*grid[i][j].x+grid[i-1][j].x;
}

double d2xdeta2( int i, int j ){
        return grid[i][j+1].x-2*grid[i][j].x+grid[i][j-1].x;
}

double d2xdxideta( int i, int j ){
        return (grid[i+1][j+1].x-grid[i+1][j-1].x-grid[i-1][j+1].x+grid[i-1][j-1].x)/4;
}

double d2ydxi2( int i, int j ){
        return grid[i+1][j].y-2*grid[i][j].y+grid[i-1][j].y;
}

double d2ydeta2( int i, int j ){
        return grid[i][j+1].y-2*grid[i][j].y+grid[i][j-1].y;
}

double d2ydxideta( int i, int j ){
```

```
        return (grid[i+1][j+1].y-grid[i+1][j-1].y-grid[i-1][j+1].y+grid[i-1][j-1].y)/4;
}

//-------------------------------------------------//

// controls
double phiBoundary( int i, int j ){
        double ans;
        if( abs(dxdxi(i,j)) > abs(dydxi(i,j)) ){
                ans = -d2xdxi2(i,j)/dxdxi(i,j);
        } else {
                ans = -d2ydxi2(i,j)/dydxi(i,j);
        }
        return ans;
}

double phi( int i, int j ){
        switch( custom ){
                case 0:
                        return 0;
                case 1: {
                        double phi0 = phiBoundary( i, 0 );
                        double phiMax = phiBoundary( i, JMAX-1 );

                        double ans = phi0 + j/(JMAX-1)*(phiMax-phi0);

                        return ans;
                }
                case 2:
                        return 0;
        }
}

double psiBoundary( int i, int j ){
        double ans;
        if( abs(dxdeta(i,j)) > abs(dydeta(i,j)) ){
                ans = -d2xdeta2(i,j)/dxdeta(i,j);
        } else {
                ans = -d2ydeta2(i,j)/dydeta(i,j);
        }
        return ans;
}

double psi( int i, int j ){
        switch( custom ){
                case 0:
                        return 0;
                case 1: {
                        double psi0 = psiBoundary( 0, j );
                        double psiMax = psiBoundary( IMAX-1, j );

                        double ans = psi0 + i/(IMAX-1)*(psiMax-psi0);

                        return ans;
```

```
                }
                case 2:
                        return -dxdxi(i,j);
        }
}

double A1( int i, int j ){
        double ans = pow( dxdeta(i,j), 2 ) + pow( dydeta(i,j), 2 );
        return ans;
}

double A2( int i, int j ){
        double ans = dxdxi(i,j)*dxdeta(i,j)+dydxi(i,j)*dydeta(i,j);
        return ans;
}

double A3( int i, int j ){
        double ans = pow( dxdxi(i,j), 2 ) + pow( dydxi(i,j), 2 );
        return ans;
}
```