Cell-Centered, Finite Volume Euler Solver using the Van Leer Flux-Splitting Method and an m-Stage Timestepping Scheme

Jafar Mohammed

AE516-01

December 8, 2008

Euler Equations

The Euler equations relate the velocity, density and pressure of a moving fluid. Unlike the Navier-Stokes equations, these equations neglect the effects of viscosity. Therefore, a solution of the Euler equations represents an approximation of the real fluids problem. The conservation of mass, momentum and energy equations are the governing set of equations that need to be solved. Thus, the invicid, compressible Euler equations, in two dimensions, is given in its combined conservation form:

$$\frac{d}{dt}\iint UdA + \oint (F\hat{\imath} + G\hat{\jmath}) \cdot \hat{n} dS = 0$$

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix} \qquad F = \begin{bmatrix} \rho u \\ \rho u^2 + P \\ \rho uv \\ \rho uH \end{bmatrix} \qquad G = \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + P \\ \rho vH \end{bmatrix}$$

where U is the conservative state vector, F is the invicid flux vector in the x-direction and G is the invicid flux vector in the y-direction.

The equations governing total energy, E, and total enthalpy, H, need to be defined:

$$E = \frac{P}{(\gamma - 1)\rho} + \frac{1}{2}(u^2 + v^2)$$

$$H = E + \frac{P}{\rho}$$

The pressure can be solved using the ideal gas law:

$$P = \rho RT = (\gamma - 1)\left[\rho E - \frac{1}{2}\rho(u^2 + v^2)\right]$$

Although the Euler equations can be solved analytically, it is preferred to solve them computationally by iterative methods. A common time-stepping algorithm is the forward Euler step, however it has timestep restrictions in order to maintain stability. During each iteration, the fluxes are calculated and added to the conservative state vector. The solution is converged when the change in flux (normalized residual) approaches zero. Different flux calculation methods and time-stepping algorithms can enhance stability and accuracy. In this project, the Van Leer flux-splitting scheme and the fourth-order Runge Kutta time-stepping algorithm was used, and will be explained henceforth.

Euler Code

The code was divided into five major sections:

- Loading/Creating a grid
- Calculating grid metrics
- Setting freestream condition and loop variables
- Initializing the solution to freestream conditions
- Executing the main loop in time
    - Calculating the residual

Loading the Grid

The code provides the option to create a Cartesian grid or load a grid in Plot3D format. When a grid is loaded, the number of zones is checked. If the number of zones is greater than one, then the grid will not be loaded because the code cannot handle multiple zones. If the grid passes this check, then the file is read and the x, y and z coordinates are stored in arrays. The code which completes this task is shown below:

```
if (zones == 1)
      % Read in number of i,j,k points
      npi = fscanf(fid, '%d', 1);
      npj = fscanf(fid, '%d', 1);
      npk = fscanf(fid, '%d', 1);

      % Retrieve i,j,k coordinates
      x = fscanf(fid, '%f', [npi,npj]);
      y = fscanf(fid, '%f', [npi,npj]);
      z = fscanf(fid, '%f', [npi,npj]);
      disp('Grid read successfully');
end
```

Calculating Grid Metrics

Once the grid is loaded, then the grid metrics can be calculated. Although it is feasible to calculate quantities during the iteration, it may be inefficient. Therefore, all necessary quantities will be calculated and stored in arrays before the iterations begin. Since this code is a cell-centered, finite volume solver, certain cell quantities will be needed such as the areas of each face (north, south, east, west), the outward normal vector of each face, and the volume of each cell.

The area of each face is calculated by taking the square root of the square of the x and y lengths of the face. For example, the east face area is calculated as follows:

$$sE = \sqrt{(x_E)^2 + (y_E)^2}$$

where sE is the area of the east face, x_E is the x length of the east face and y_E is the y length of the east face.

The volume of each cell is calculated using the formula for the volume of a parallelepiped (or the absolute value of the scalar triple product), which can be found in any calculus book.

The normal vector for each face was found by crossing the unit normal vector of the face by the unit vector in the z-direction. The result is an outward normal of each face.

The code which accomplishes these three tasks is shown below:

```
% Compute volume of cell using A.BxC (volume of parallelepiped)
volume(i,j) = abs(dot(z_width,cross(-1*[s_xlen,s_ylen,0],...
                      [e_xlen,e_ylen,0])));

% Compute area of cell
sE(i,j) = sqrt((e_xlen)^2 + (e_ylen)^2);
sN(i,j) = sqrt((n_xlen)^2 + (n_ylen)^2);
sW(i,j) = sqrt((w_xlen)^2 + (w_ylen)^2);
sS(i,j) = sqrt((s_xlen)^2 + (s_ylen)^2);

% Compute outward normal of faces (return 3 component vector)
temp_nE = cross([e_xlen,e_ylen, 0]/sE(i,j), z_width);
temp_nN = cross([n_xlen,n_ylen, 0]/sN(i,j), z_width);
temp_nW = cross([w_xlen,w_ylen, 0]/sW(i,j), z_width);
temp_nS = cross([s_xlen,s_ylen, 0]/sS(i,j), z_width);
```

Setting Freestream Conditions and Loop Variables

Before the solution can be initialized, the freestream conditions need to be specified. The following freestream quantities are given:

- Density: 1.2 kg/m^2
- Temperature: 300 K
- Pressure: 100,000 Pa
- Mach: 0.3, 0.7, 1.5 (depending on test case)

After the freestream conditions are defined, the speed of sound is calculated and used to find the u and v velocities. The primitive state vector, V, contains the primitive variables of the solution. In this code:

$$V = \begin{bmatrix} \rho \\ u \\ v \\ P \end{bmatrix}$$

where ρ is the density, u is the velocity in the x-dir., v is the velocity in the y-dir., and P is the pressure.

Also, several looping variable are set. The number of iterations and the frequency of text output can are defined. The user can choose between local and global timestepping. If global timestepping is chosen, then a global timestep must be set. If local timestepping is chosen, then a CFL number must be set. The user also has the ability to plot contours of the primitive variables, which are updated during each iteration (or at a specified frequency).

This code employs the m-stage timestepping scheme, as outlined by Jameson. This allows the user to choose the order of the timestepping scheme. For example, if the order of the scheme is 1, then the forward Euler step is used. If the order of the scheme is 4, then a modified fourth-order Runge Kutta timestepping scheme is used. However, for non-linear equations, the modified fourth-order Runge Kutta scheme is only second-order accurate. The fourth-order scheme enables the user to use a higher CFL number and less iterations to achieve an accurate result.

Initializing the Solution

Initializing the solution is a short task. Each cell in the grid is assigned the freestream primitive state vector, which is then converted to the conservative state vector via a function. Also, the residual of the cell is initialized to 0. The *V*, *U*, and *resid* variables are cell arrays. This is accomplished using the following code:

```
% Loop through all cells and init PSV to freestream conditions
% Convert PSV to conservative state vector
% Init residual to 0
for i = 1:nci
    for j = 1:ncj
        V{i,j}     = V_free;
        U{i,j}     = convV_U(V{i,j});
        resid{i,j} = [0 0 0 0];
    end
end
```

Iterating the Main Loop in Time

The bulk of computing time of the code takes place in this section. The solution is iterated for a user-specified amount of time. In each iteration, the residual for each cell is calculated, and the conservative and primitive state vectors are updated – sometimes four times per iteration depending on the timestepping scheme. Also, if local timestepping is chosen, the minimum timestep allowed for each cell is calculated and used.

The following code illustrates how the solution evolves inside the main iteration loop:

```
    % Save CSV from this timestep (to be used in m-stage)
    U0 = U;

    % M-stage timestepping scheme
    for m = 1:m_stage
        % Calculate residual using function calcResid
        % Passes PSV, normals, areas cell array,
        %        freestream PSV, and nci and ncj
        resid = calcResid(V, V_free, normals, areas, nci, ncj);

        % Loop through all cells to update solution
        for i = 1:nci
            for j = 1:ncj

                % If local timestepping, calculate timestep of the cell
                if gbl_timestep == 0
                    vel = [V{i,j}(2) V{i,j}(3)];
                    cell_a = speedsound(V{i,j}(4),V{i,j}(1));
                    dt(1) = CFL * sE(i,j)/(abs(vel(1)*nE{i,j}(1) +...
                            vel(2)*nE{i,j}(2))+cell_a);
                    dt(2) = CFL * sN(i,j)/(abs(vel(1)*nN{i,j}(1) +...
                            vel(2)*nN{i,j}(2))+cell_a);
                    dt(3) = CFL * sW(i,j)/(abs(vel(1)*nW{i,j}(1) +...
                            vel(2)*nW{i,j}(2))+cell_a);
                    dt(4) = CFL * sS(i,j)/(abs(vel(1)*nS{i,j}(1) +...
                            vel(2)*nS{i,j}(2))+cell_a);
                    timestep = min(dt);
                end

                % Update solution using the saved CSV
                % Multiply by 'alpha' constant
                U{i,j} = U0{i,j} -...
                         1/(m_stage-(m-1))*timestep/volume(i,j)*resid{i,j};

                % Update cell PSV
                V{i,j} = convU_V(U{i,j});
            end
        end
    end
```

First, the conservative state vector cell array of the current iteration is saved because it will be used in the m-stage timestepping scheme. The code then enters the m-stage loop, which is an inner-iteration loop. At this point the residual is calculated. The details of this will be presented in the next section.

After the residual is calculated, the code enters two loops (loops in the i and j directions), which set up updating the solution. The local timestep of each cell is then calculated by taking the minimum timestep for a given face. For the east face, the timestep calculation is as follows:

$$dt = CFL\left(\frac{sE}{|\bar{V} \times \hat{n}E| + a}\right)$$

where CFL is the CFL number, sE is the area of the east face, $\bar{V}$ is the velocity vector, $\hat{n}E$ is the outward normal vector for the east face, and a is the speed of sound of the cell.

The solution of a cell is then updated by the following equation:

$$U = U_0 + \alpha_m \frac{\Delta t}{V} \mathcal{R}$$

where U_0 is the saved conservative state vector, Δt is the timestep, V is the volume of the cell, and $\mathcal{R}$ is the residual.

α_m is a constant which depends on the iteration of the m-stage loop. If the m-stage loop is first order (m-stage = 1), then α_m = 1. However, if m-stage = 4, then α_m is $1/4$ for the first iteration, $1/3$ for the second iteration, $1/2$ for the third iteration, and 1 for the fourth iteration. Since the order of the m-stage loop is set beforehand (in the variable *m_stage*) and the current place in the iteration is known (given by variable m), then an equation for α_m can be developed:

$$\alpha_m = \frac{1}{m - (1 - m\_stage)}$$

After the solution is updated, the primitive state vector for each cell is calculated from the updated conservative state vector, and the process repeats itself for each iteration of the m-stage loop, and then each iteration of the main loop.

Calculating the Residual

The method of calculating the residual is the most important part of the code. Many schemes exist, however, for this project the first-order upwind Van Leer flux-splitting scheme was used because it was the most straightforward in implementation.

For $-1 < M < 1$, the flux is split into right and left components, as follows:

$$F^{\pm} = \pm\frac{\rho a}{4}\left(\widehat{M} + 1\right)\begin{bmatrix} 1 \\ u + \dfrac{n_x(-\hat{u} \pm 2a)}{\gamma} \\ v + \dfrac{n_y(-\hat{u} \pm 2a)}{\gamma} \\ h_0 + \dfrac{a^2\left(\widehat{M} \mp 1\right)}{\gamma + 1} \end{bmatrix}$$

where ρ is the density of the cell, a is the speed of sound of the cell, $\widehat{M}$ is the contravariant Mach number of the cell face, u & v are the velocities in the x and y directions, n_x & n_y are the outward normals in the x and y directions of the cell face, $\hat{u}$ is the contravariant velocity of the cell face, h_0 is the total enthalpy of the cell, and γ is the specific-heat ratio.

Therefore, the flux for the east face of the cell(i,j), would be

$$F_E = F^{+}_{i,j} + F^{-}_{i+1,j}$$

For the north face, the flux would be

$$F_N = F^{+}_{i,j} + F^{-}_{i,j+1}$$

In order to preserve the upwinding characteristic of the scheme, it must be enforced that the flux of the west face of a cell equal the flux of the east face of the adjacent cell (i-1,j), accounting for the change in sign of the normal vector. The same is true for the south face of a cell. Therefore the flux for the west and south faces are as follows:

$$F_W = -F_{E\,i-1,j}$$

$$F_S = -F_{N\,i,j-1}$$

If $M \geq 1$, then the flux of the face is equal to the full flux vector:

$$\oint (F\hat{\imath} + G\hat{\jmath}) \cdot \hat{n} dS$$

In this code, the north and south boundaries of the grid are treated as invicid walls, the west face is treated as an inflow boundary and the east face is an outflow boundary.

For the invicid walls, the component of the velocity normal to the wall must be removed. The following equation is used to accomplish this task (for the south boundary):

$$\overline{V_b} = \bar{V}_{i,1} - \left(\bar{V}_{i,1} \cdot \hat{n}_b\right)\hat{n}_b$$

where $\overline{V_b}$ is the velocity vector of the boundary, $\bar{V}_{i,1}$ is the velocity vector of the cell adjacent to the wall, and $\hat{n}_b$ is the normal vector of the wall (which is the outward normal vector of the cell face adjacent to the wall).

For the subsonic inflow boundary, the density and pressure were specified as their freestream counterparts, while the *u* and *v* velocities were extrapolated using the Riemann invariants. For the subsonic outflow boundary, the *u* and *v* velocities were also extrapolated using the Riemann invariants, while the density and pressure were assigned the value of the cell adjacent to the boundary.

For the supersonic inflow boundary, freestream conditions were applied directly. And for the supersonic outflow boundary, the density, pressure and velocities were taken from the cell adjacent to the boundary.

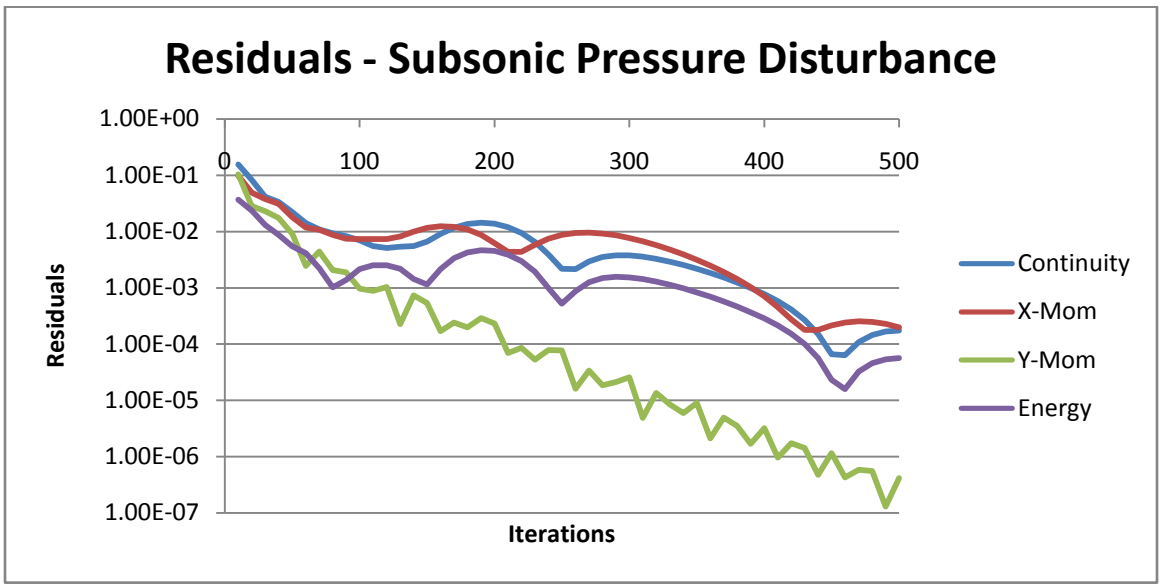
After all of the face fluxes were calculated, the residual of the cell was found using the following equation:

$$\mathcal{R} = F_E s_E + F_N s_N + F_W s_W + F_S s_S$$

where *F* is the flux of the specified face and *s* is the area of that face.

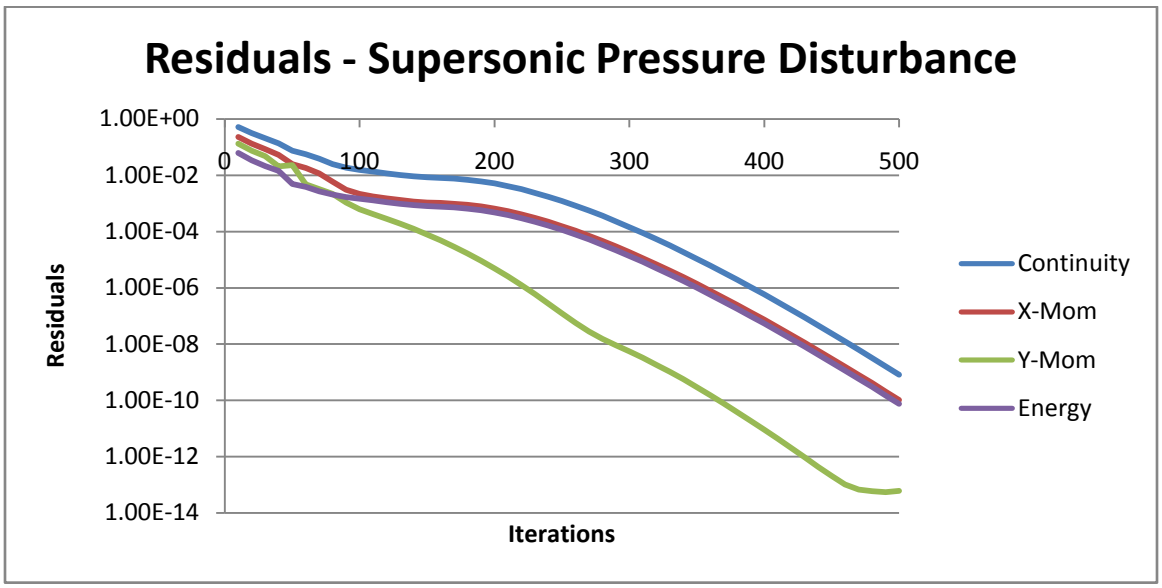
Results for Disturbance Test at M=0.3

- The grid was equally spaced with 20 cells in the i-direction and 10 cells in the j-direction.
- A pressure disturbance was applied to cell (10,5) of 1.1*(Freestream Pressure).
- Disturbance dissipated after 500 iterations.
- Video provided of disturbance dissipation



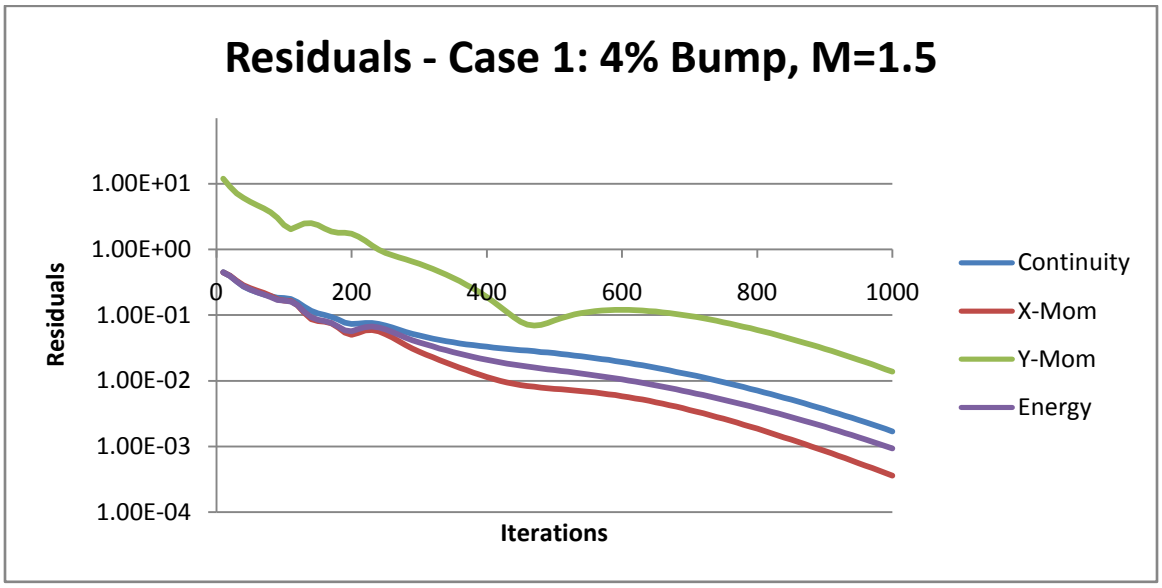
Results for Disturbance Test at M=1.5

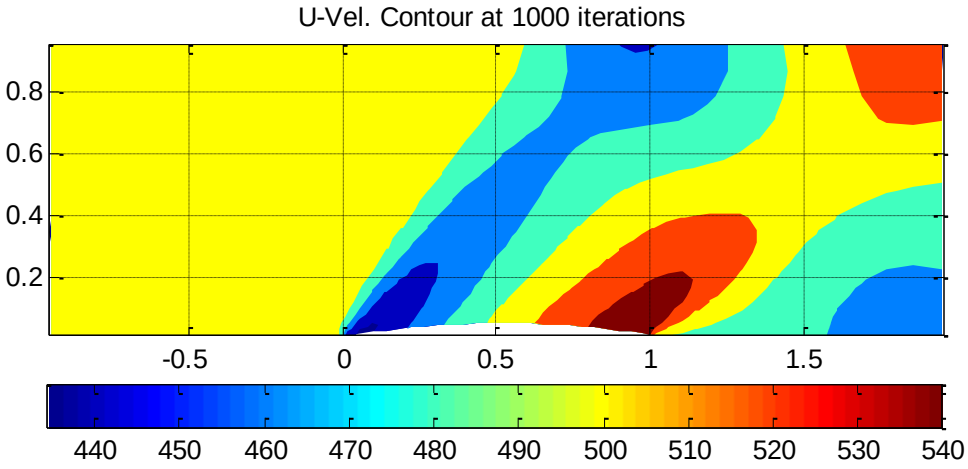
- The grid was equally spaced with 20 cells in the i-direction and 10 cells in the j-direction.
- A pressure disturbance was applied to cell (10,5) of 1.1*(Freestream Pressure).
- Disturbance dissipated in less than 500 iterations.
- Video provided of disturbance dissipation

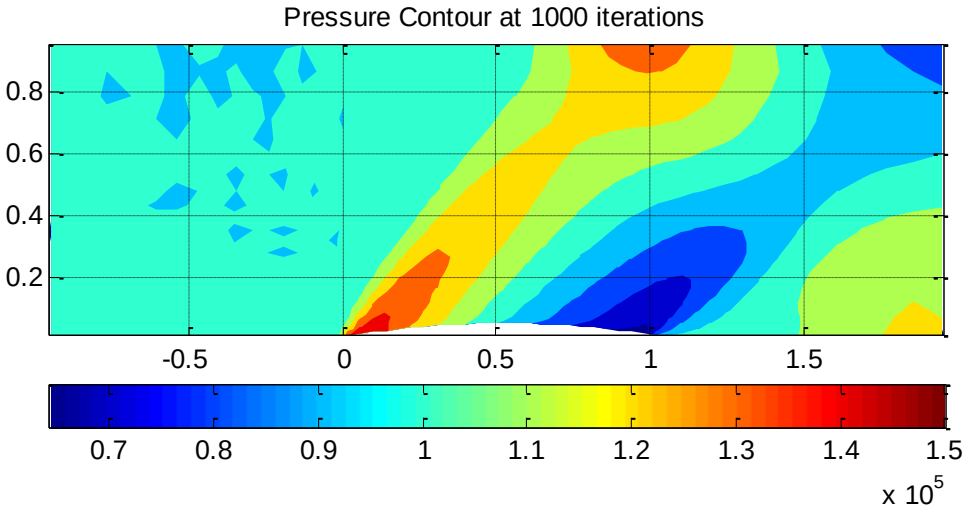


Results for Case 1 (Bump Grid, M = 1.5)

- Grid: bump04perc.grd
- Solver run for 1000 iterations (Time: 4096 sec)
    - CFL = 0.5
    - Fourth-Order modified Runge-Kutta

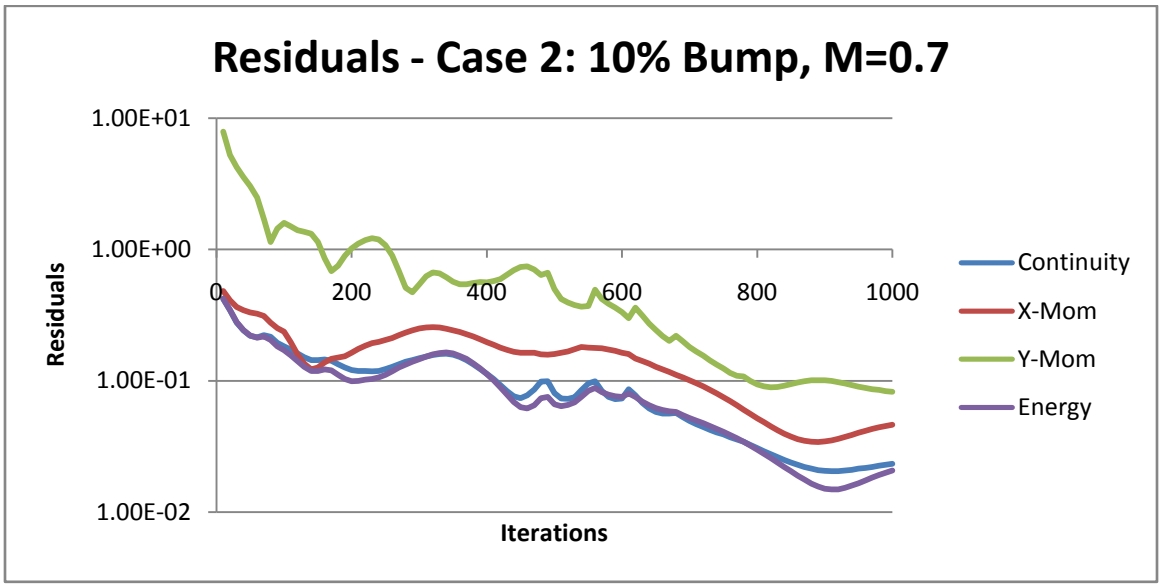


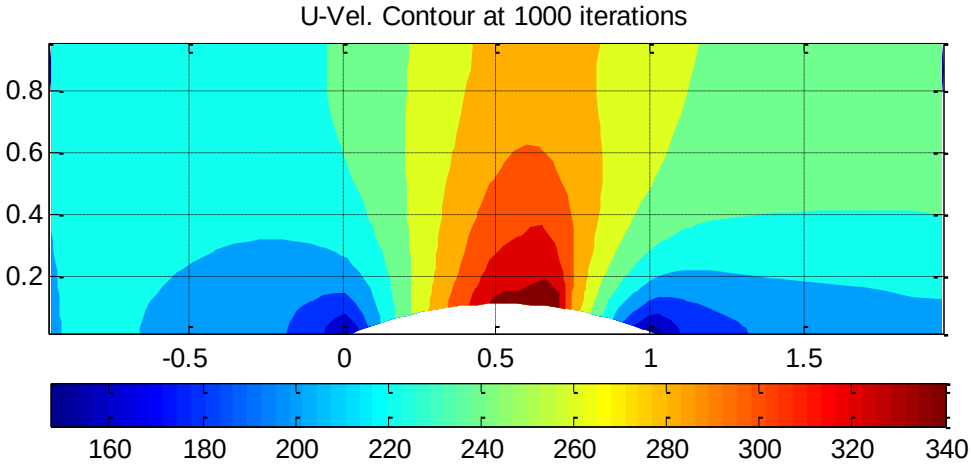

U-Vel. Contour at 1000 iterations
0.8
0.6
0.4
0.2
-0.5
0
0.5
1
1.5
440
450
460
470
480
490
500
510
520
530
540

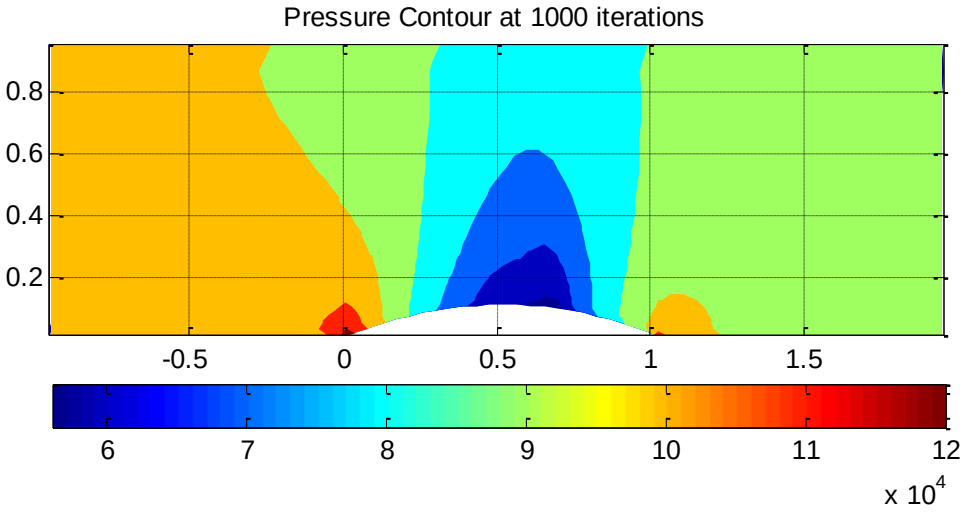

Pressure Contour at 1000 iterations
0.8
0.6
0.4
0.2
-0.5
0
0.5
1
1.5
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4
1.5
x 10^5

Results for Case 2 (Bump Grid, M = 0.7)

- Grid: bump10perc.grd
- Solver run for 1000 iterations (Time: 3236 sec)
    - CFL = 0.5
    - Fourth-Order modified Runge-Kutta

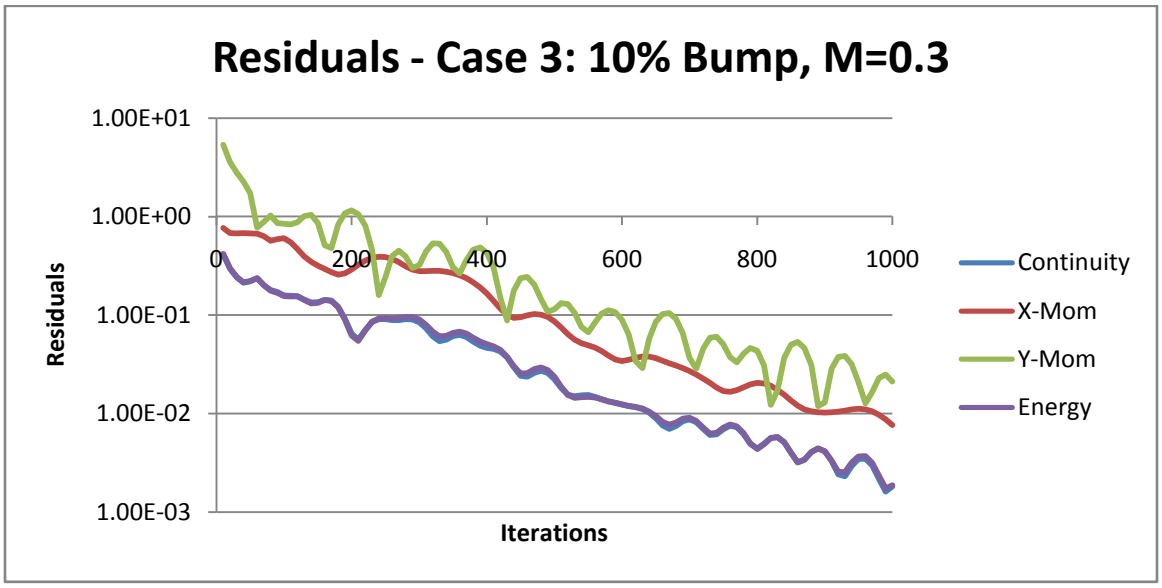


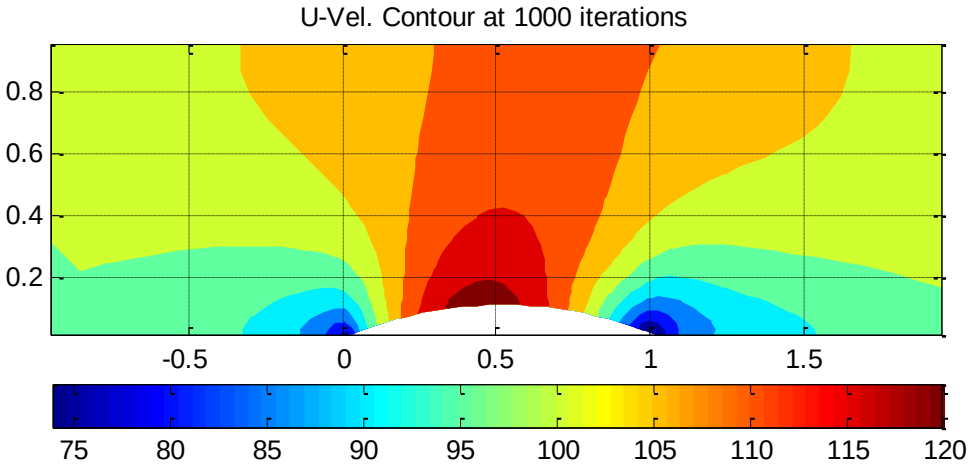

U-Vel. Contour at 1000 iterations
0.8
0.6
0.4
0.2
-0.5
0
0.5
1
1.5
160
180
200
220
240
260
280
300
320
340

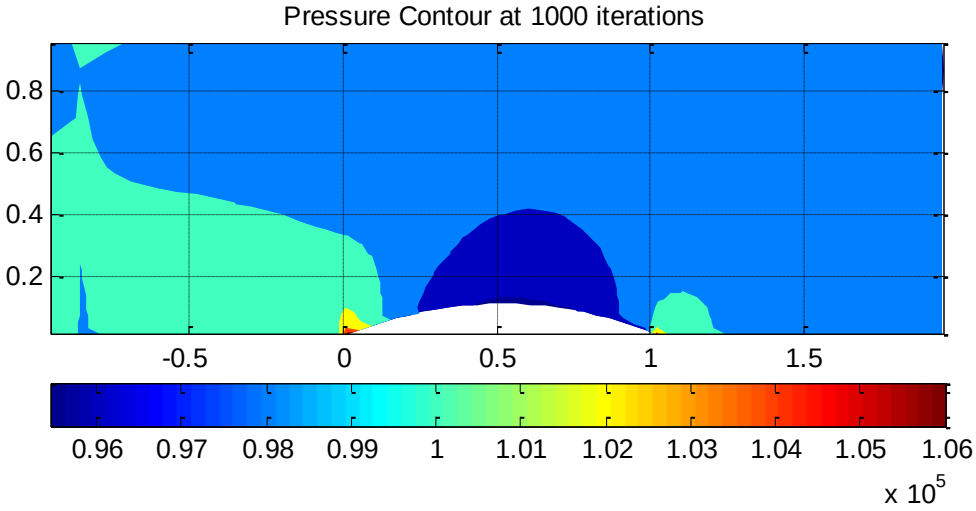

Pressure Contour at 1000 iterations
0.8
0.6
0.4
0.2
-0.5
0
0.5
1
1.5
6
7
8
9
10
11
12
x 10^4

Results for Case 3 (Bump Grid, M = 0.3)

- Grid: bump10perc.grd
- Solver run for 1000 iterations (Time: 4609 sec)
    - CFL = 0.5
    - Fourth-Order modified Runge-Kutta

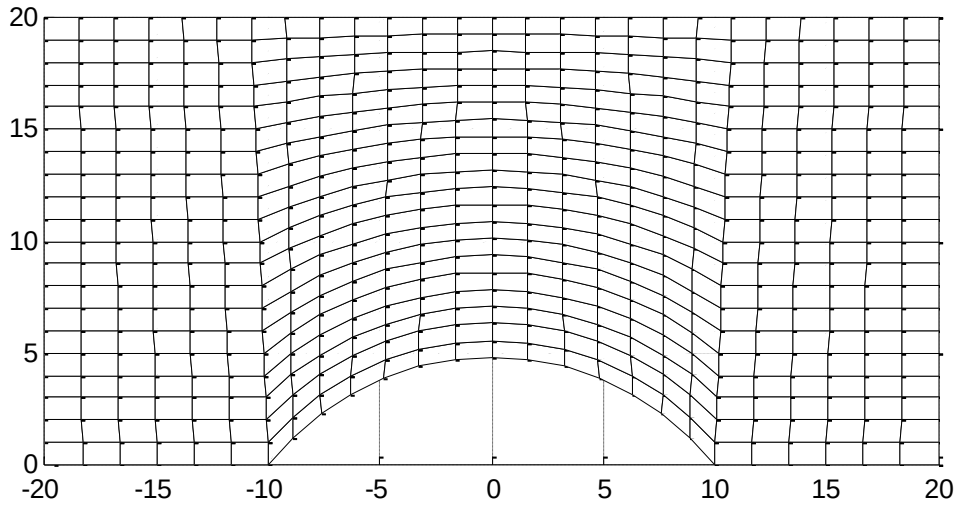



U-Vel. Contour at 1000 iterations
0.8
0.6
0.4
0.2
-0.5
0
0.5
1
1.5
75
80
85
90
95
100
105
110
115
120

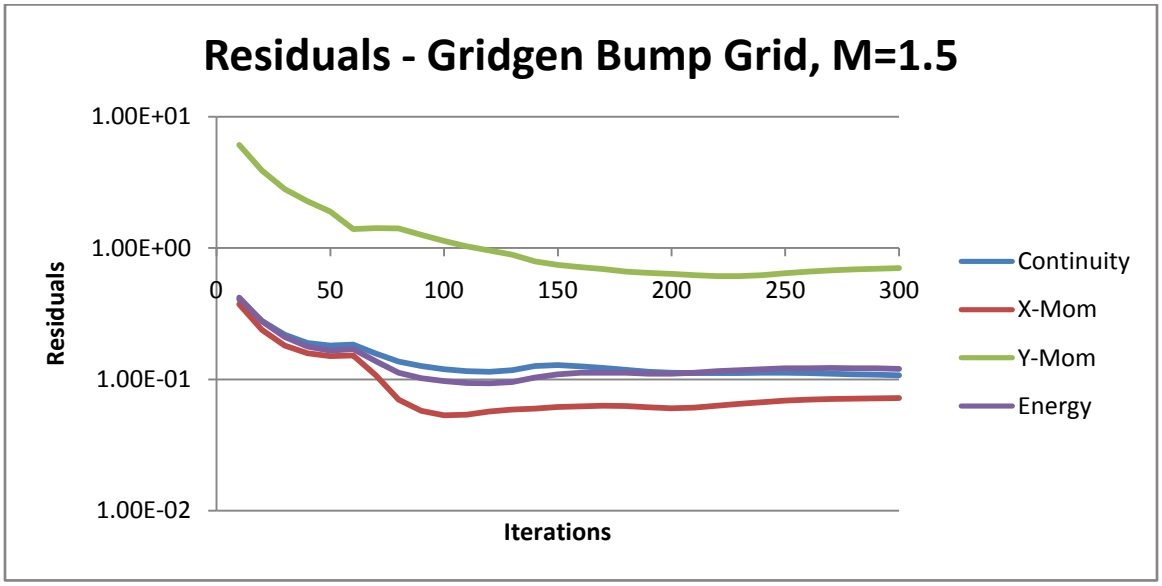

Pressure Contour at 1000 iterations
0.8
0.6
0.4
0.2
-0.5
0
0.5
1
1.5
0.96
0.97
0.98
0.99
1
1.01
1.02
1.03
1.04
1.05
1.06
x 10^5

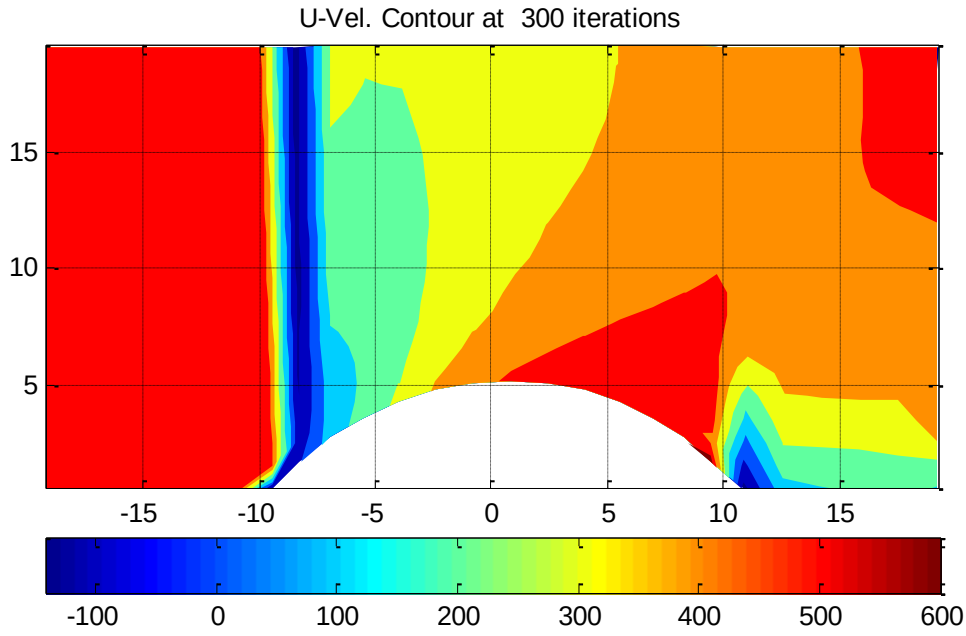
Additional: Bump Grid provided by Gridgen

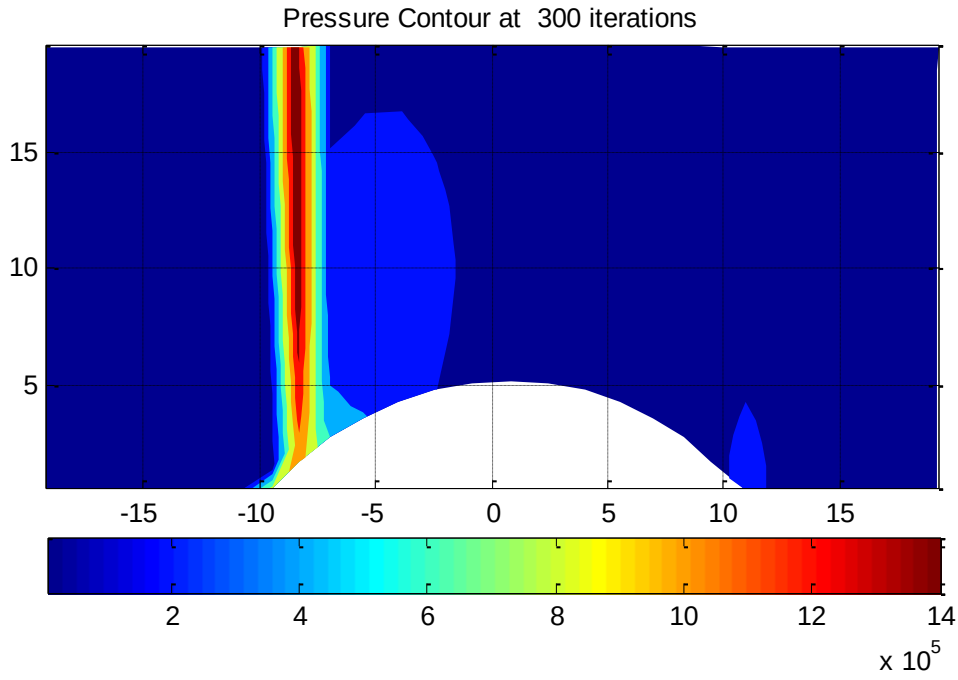
- Mach = 1.5
- Solver run for 300 iterations (Time: 366 sec)
    - CFL = 0.7
    - Fourth-Order modified Runge-Kutta
- Video provided



Grid provided by Gridgen




U-Vel. Contour at 300 iterations
15
10
5
-15
-10
-5
0
5
10
15
-100
0
100
200
300
400
500
600


Pressure Contour at 300 iterations
15
10
5
-15
-10
-5
0
5
10
15
2
4
6
8
10
12
14
x 10^5

Source Code

- euler.m – main function
- calcResid.m – calculates residual
- flux.m – calculates full flux vector
- fpos.m – calculates F+ for Van Leer
- fneg.m – calculates F- for Van Leer
- e_0.m – calculates total energy
- h_0.m – calculates total enthalpy
- convU_V.m – converts conservative to primitive state vector
- convV_U.m – converts primitive to conservative state vector
- speedsound.m – calculates the speed of sound
- fixTime.m – generates time left

euler.m

```
% Euler Solver
% First Order Van Leer Scheme
% M-stage timestepping

clear                                % Clear variables from memory
clc                                  % Clear the command window

%% LOAD/GENERATE GRID %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
loadGrid = 1;            %0 - create cartesian grid
                         %1 - load grid from file
tic
if loadGrid == 1
    % Open the file and assign it an ID
    %fid = fopen('BUMP/bump10perc.grd', 'r');
    fid = fopen('BUMP/bump04perc.grd', 'r');
    %fid = fopen('bump.grd', 'r');

    if fid >= 1
        % Read in file headers
        zones = fscanf(fid, '%d', 1);

        % Code only handles 1 zone
        % Therefore, check for number of zones
        if (zones == 1)
            % Read in number of i,j,k points
            npi = fscanf(fid, '%d', 1);
            npj = fscanf(fid, '%d', 1);
            npk = fscanf(fid, '%d', 1);

            % Retrieve i,j,k coordinates
            x = fscanf(fid, '%f', [npi,npj]);
            y = fscanf(fid, '%f', [npi,npj]);
            z = fscanf(fid, '%f', [npi,npj]);
            disp('Grid read successfully');
        end
        fclose(fid);
    end
else
    % Generate cartesian grid using the meshgrid function
    npi    = 21;                         % Num. of pts. in i-dir
    npj    = 11;                         % Num. of pts. in j-dir
    xc     = linspace(0,20,npi);         % Distribution in i-dir
    yc     = linspace(0,10,npj);         % Distribution in j-dir
    [y,x] = meshgrid(yc,xc);             % Creates 2D grid
end

% Visualizes the mesh in a plot window
%mesh(x,y,zeros(npi,npj))
%axis image;view(2);drawnow;

disp('Grid generated');
toc;disp(' ');
```

```
%% GRID METRICS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic
z_width = [0,0,1];                      % Unit vector in z-dir
nci = npi-1;                            % Number of cells (pts-1) in i dir
ncj = npj-1;                            % Number of cells (pts-1) in j dir

for i = 1:nci
    for j = 1:ncj
        % Assemble the face lengths
        e_xlen = x(i+1,j+1)-x(i+1,j);
        e_ylen = y(i+1,j+1)-y(i+1,j);

        n_xlen = x(i,j+1)-x(i+1,j+1);
        n_ylen = y(i,j+1)-y(i+1,j+1);

        w_xlen = x(i,j)-x(i,j+1);
        w_ylen = y(i,j)-y(i,j+1);

        s_xlen = x(i+1,j)-x(i,j);
        s_ylen = y(i+1,j)-y(i,j);

        % Compute midpoint of cell (for plotting)
        xmid(i,j) = (x(i,j) + x(i+1,j))/2;
        ymid(i,j) = (y(i,j) + y(i,j+1))/2;

        % Compute volume of cell using A.BxC (volume of parallelepiped)
        volume(i,j) = abs(dot(z_width,cross(-1*[s_xlen,s_ylen,0],...
                      [e_xlen,e_ylen,0])));

        % Compute area of cell
        sE(i,j) = sqrt((e_xlen)^2 + (e_ylen)^2);
        sN(i,j) = sqrt((n_xlen)^2 + (n_ylen)^2);
        sW(i,j) = sqrt((w_xlen)^2 + (w_ylen)^2);
        sS(i,j) = sqrt((s_xlen)^2 + (s_ylen)^2);

        % Compute outward normal of faces (return 3 component vector)
        temp_nE = cross([e_xlen,e_ylen, 0]/sE(i,j), z_width);
        temp_nN = cross([n_xlen,n_ylen, 0]/sN(i,j), z_width);
        temp_nW = cross([w_xlen,w_ylen, 0]/sW(i,j), z_width);
        temp_nS = cross([s_xlen,s_ylen, 0]/sS(i,j), z_width);

        % Truncate normal vector to 2 components
        nE{i,j} = [temp_nE(1) temp_nE(2)];
        nN{i,j} = [temp_nN(1) temp_nN(2)];
        nW{i,j} = [temp_nW(1) temp_nW(2)];
        nS{i,j} = [temp_nS(1) temp_nS(2)];

        % Clear unecessary variables
        clear temp_nE temp_nN temp_nW temp_nS
        clear e_xlen n_xlen w_xlen s_xlen
        clear e_ylen n_ylen w_ylen s_ylen
    end
end
disp('Grid Metrics calculated');
```

```
toc;disp(' ');

%% SET FREESTREAM CONDITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
free_rho      = 1.2;                          % kg/m^3
free_T        = 300.00;                       % Kelvin
free_P        = 100000;                       % Pa
free_M        = 1.5;                          % Mach
free_aoa      = 0;                            % deg

free_a        = speedsound(free_P,free_rho);  % m/s
free_u        = free_M*free_a*cosd(free_aoa); % m/s
free_v        = free_M*free_a*sind(free_aoa); % m/s
free_vel      = [free_u free_v];

% Freestream Primitive State Vector (PSV)
V_free        = [free_rho free_u free_v free_P];

diary('output.txt')
fprintf('Freestream Conditions:\n');
fprintf('Mach:         %10.2f\n',free_M);
fprintf('Flow AOA:     %10.2f deg\n',free_aoa);
fprintf('u Velocity:   %10.2f m/s\n',free_u);
fprintf('v Velocity:   %10.2f m/s\n',free_v);
fprintf('Pressure:     %10.2e Pa\n',free_P);
fprintf('Temperature:  %10.2f K\n',free_T);
fprintf('Density:      %10.2f kg/m^3\n\n',free_rho);

%% SET ITERATION VARIABLES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Iteration variables
iterations   = 100;             % Number of iterations to run

gbl_timestep = 0;               % 0 = local timestepping
                                % 1 = global timestepping

timestep     = 1e-5;            % Timestep for global timestepping
CFL          = 0.5;             % Courant number

m_stage      = 4;               % m-stage time stepping
                                % e.g. 1 for Euler step
                                %      4 for 4th-order RK

% Output variables
freq         = 10;              % Reporting frequency for output

plotcontours = 0;               % Plot contours during iterations
                                % 0 - off
                                % 1 - Density
                                % 2 - U Velocity
                                % 3 - V Velocity
                                % 4 - Pressure

plots_on     = 1 ;              % Plot PSV contour plots after
                                % completion
```

```
fprintf('Iteration Variables:\n');
fprintf('Iterations:   %5d\n',iterations);
fprintf('CFL:          %5.2f\n',CFL);
fprintf('M-Stage:      %5d\n',m_stage);
if gbl_timestep == 0
    fprintf('Timestepping  %5s\n\n','local');
else
    fprintf('Timestepping %5s\n\n','global');
end

%% INITIALIZE SOLUTION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic
resid_i      = 0;               % Iterative residual
resid_0      = 0;               % Step 0 residual
start_iter   = 0;               % Used for multiple runs
end_iter     = 0;               % Used for multiple runs
residReduced = 0;               % If divergence detected
fm = 1;                         % Used for capturing movies

% Combine normals and areas into big cell array which will be passed
% to the function which computes the residual
normals = {nE nN nW nS};
areas   = {sE sN sW sS};

% Initalize variables which will allow for visualization
% i.e. Plot Contours
con_density = zeros([nci,ncj]);
con_uvel    = zeros([nci,ncj]);
con_vvel    = zeros([nci,ncj]);
con_pres    = zeros([nci,ncj]);

% Loop through all cells and init PSV to freestream conditions
% Convert PSV to conservative state vector
% Init residual to 0
for i = 1:nci
    for j = 1:ncj
        V{i,j}     = V_free;
        U{i,j}     = convV_U(V{i,j});
        resid{i,j} = [0 0 0 0];
    end
end

% Add a disturbance at cell (10,5) if cartesian grid created
if loadGrid == 0
    V{10,5}(4) = 1.1*free_P;
    U{10,5}    = convV_U(V{10,5});
end

diary off
disp('Solution initialized');
toc

%% MAIN LOOP %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
start_iter = start_iter + 1;        % Start at iteration 1
diary on
```

```
% Create plot window
if plotcontours == 1
    hc = figure('name','Density Contour');
elseif plotcontours == 2
    hc = figure('name','U Velocity Contour');
elseif plotcontours == 3
    hc = figure('name','V Velocity Contour');
elseif plotcontours == 4
    hc = figure('name','Pressure Contour');
end

tic

% Main loop in time
for iter = start_iter:(end_iter + iterations)
    % Time variable used to measure time/iteration
    ti1 = cputime;

    % Initialize iteration residual to 0
    resid_i = 0;

    % Save CSV from this timestep (to be used in m-stage)
    U0 = U;

    % M-stage timestepping scheme
    for m = 1:m_stage
        % Calculate residual using function calcResid
        % Passes PSV, normals, areas cell array,
        %        freestream PSV, and nci and ncj
        resid = calcResid(V, V_free, normals, areas, nci, ncj);

        % Loop through all cells to update solution
        for i = 1:nci
            for j = 1:ncj

                % If local timestepping, calculate timestep of the cell
                if gbl_timestep == 0
                    vel = [V{i,j}(2) V{i,j}(3)];
                    cell_a = speedsound(V{i,j}(4),V{i,j}(1));
                    dt(1) = CFL * sE(i,j)/(abs(vel(1)*nE{i,j}(1) +...
                            vel(2)*nE{i,j}(2))+cell_a);
                    dt(2) = CFL * sN(i,j)/(abs(vel(1)*nN{i,j}(1) +...
                            vel(2)*nN{i,j}(2))+cell_a);
                    dt(3) = CFL * sW(i,j)/(abs(vel(1)*nW{i,j}(1) +...
                            vel(2)*nW{i,j}(2))+cell_a);
                    dt(4) = CFL * sS(i,j)/(abs(vel(1)*nS{i,j}(1) +...
                            vel(2)*nS{i,j}(2))+cell_a);
                    timestep = min(dt);
                end

                % Update solution using the saved CSV
                % Multiply by 'alpha' constant
                U{i,j} = U0{i,j} -...
                         1/(m_stage-(m-1))*timestep/volume(i,j)*resid{i,j};
```

```
                % Update cell PSV
                V{i,j} = convU_V(U{i,j});

                % Update contour arrays used for plotting
                con_density(i,j) = V{i,j}(1);
                con_uvel(i,j)    = V{i,j}(2);
                con_vvel(i,j)    = V{i,j}(3);
                con_pres(i,j)    = V{i,j}(4);

                % Assemble first part of L2 norm for residual
                resid_i = resid_i + resid{i,j}.^2;
            end
        end
    end

    % Assemble second part of L2 norm for residual
    resid_i = (resid_i).^.5/(nci*ncj);

    % Assign normalization value in first interation
    if iter == 1
        resid_0 = resid_i;

        % More detailed iurput
        fprintf('\nIter  cont. resid  x-mom resid  y-mom resid  energy resid  Time
left\n');
    end

    % Detects extreme divergence (at the point of no return)
    % and shuts down simulation
    if isnan(resid_i/resid_0)
        break;
        disp('Solution corrupt.');
    end

    % Detects divergence happening in x-mom resid and cuts CFL in half
    if ((resid_i(2)/resid_0(2)) >= (1e1))
        if residReduced == 0
            CFL = CFL/2;
            notice = sprintf('Divergence detected.  CFL reduced to %5.2f',CFL);
            disp(notice);
            residReduced = residReduced + 1;
        %elseif ((residReduced > 0) &&((resid_i(2)/resid_0(2)) >= (2e1)))
        %    CFL = CFL/2;
        %    notice = sprintf('Divergence detected.  CFL reduced to %5.2f',CFL);
        %    disp(notice);
        end
    end

    % Computes time/iteration
    ti2 = cputime-ti1;

    % Displays output and updates plots at user specified time
    % interval
    if mod(iter,freq) == 0
```

```
        % Plot contours if wanted
        if plotcontours == 1
            % Assembles contour plot
            [C,h] = contourf(xmid',ymid',con_density');

            % Removes lines from contour plot
            set(h, 'LineStyle','none');

            % Adds a title to the plot
            s = sprintf('Density Contour at %4d iterations',iter);
            title(s)

            % Adds gridlines, corrects aspect ratio
            grid on;axis image;drawnow;

            % Assembles array for movie viewing
            mov(fm) = getframe(gca);
            fm=fm+1;
        elseif plotcontours == 2
            [C,h] = contourf(xmid',ymid',con_uvel');
            set(h, 'LineStyle','none');
            s = sprintf('U-Vel. Contour at %4d iterations',iter);
            title(s)
            colorbar('peer',gca,'SouthOutside');
            grid on;axis image;drawnow;
            mov(fm) = getframe(gca);
            fm=fm+1;
        elseif plotcontours == 3
            [C,h] = contourf(xmid',ymid',con_vvel');
            set(h, 'LineStyle','none');
            s = sprintf('V-Vel. Contour at %4d iterations',iter);
            title(s)
            colorbar('peer',gca,'SouthOutside');
            grid on;axis image;drawnow;
            mov(fm) = getframe(gca);
            fm=fm+1;
        elseif plotcontours == 4
            [C,h] = contourf(xmid',ymid',con_pres');
            set(h, 'LineStyle','none');
            s = sprintf('Pressure Contour at %4d iterations',iter);
            title(s)
            colorbar('peer',gca,'SouthOutside');
            grid on;axis image;drawnow;
            mov(fm) = getframe(gcf);
            fm=fm+1;
        end

        % More detailed output
        fprintf('%4d  %11.2e  %11.2e  %11.2e   %11.2e  %7s\n',iter,...
                resid_i(1)/resid_0(1),...
                resid_i(2)/resid_0(2),...
                resid_i(3)/resid_0(3),...
                resid_i(4)/resid_0(4),...
                fixTime(ti2*(end_iter+iterations-iter)));
    end
end
```

```
start_iter = iter;
end_iter = iter;
toc
diary off

%% PLOT PRIMITIVE STATE VECTOR CONTOURS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Creates a figure with 4 contour subplots
% Plot 1: Density (kg/m^3) contour
% Plot 2: U Velocity (m/s) contour
% Plot 3: V Velocity (m/s) contour
% Plot 4: Pressure (Pa)

if plots_on == 1
    figure('name','Primitive State Variables');

    subplot(221);
    contourf(xmid',ymid',con_density');
    axis image;
    colorbar('peer',gca,'SouthOutside');
    title('Density (kg/m^3)')

    subplot(222);
    contourf(xmid',ymid',con_uvel');
    axis image;
    colorbar('peer',gca,'SouthOutside');
    title('U Velocity (m/s)')

    subplot(223);
    contourf(xmid',ymid',con_vvel');
    axis image;
    colorbar('peer',gca,'SouthOutside');
    title('V Velocity (m/s)')

    subplot(224);
    contourf(xmid',ymid',con_pres');
    axis image;
    colorbar('peer',gca,'SouthOutside');
    title('Pressure (Pa)')
end

% Plots movie in new figure window
%figure;
%movie(mov);
%movie2avi(mov,'movie.avi','fps',5)
```

calcResid.m

```
%% CALCULATE RESIDUAL OF GRID %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function resid = calcResid(V, V_free, normals, areas, nci, ncj)
% Inputs: V       - Cell array containing the primitive state vector for
%                   each cell in the grid
%         V_free  - Primitive state vector containing freestream conditions
%         normals - Cell array containing all of the face normals for each
%                   cell
%         areas   - Cell array containing all of the face areas for each
%                   cell
%         nci,ncj - Number of cell in i and j

% Parse out normal vectors in smaller, specialized cell arrays
nE = normals{1};
nN = normals{2};
nW = normals{3};
nS = normals{4};

% Parse out areas vectors in smaller, specialized cell arrays
sE = areas{1};
sN = areas{2};
sW = areas{3};
sS = areas{4};

% Initialize cell array containing the flux for each face
e_flux = cell(nci,ncj);
w_flux = cell(nci,ncj);
n_flux = cell(nci,ncj);
s_flux = cell(nci,ncj);

% Initialize residual cell array
resid = cell(nci,ncj);

% Assemble freestream velocity vector
free_vel = [V_free(2) V_free(3)];

% Calculate freestream Mach number
free_a   = speedsound(V_free(4), V_free(1));

% Loop through all cells
for i = 1:nci
    for j = 1:ncj
        % Calculate speed of sound for cell
        cell_a = speedsound(V{i,j}(4),V{i,j}(1));

        % Assemble velocity vector of cell
        vel    = [V{i,j}(2) V{i,j}(3)];

        % Contravariant Mach for face
        %conM = dot(vel,nE{i,j})/cell_a;
        conM = (vel(1)*nE{i,j}(1) + vel(2)*nE{i,j}(2))/cell_a;

        % If east face is at the outflow boundary
```

```
if i == nci
    % Boundary normal equal to face normal
    nB = nE{i,j};

    % Check to see is contravariant M >= 1
    if conM >= 1
        % Assign boundary flux to nci cell flux
        Vb = V{i,j};

        % Calculate east face flux
        e_flux{i,j} = flux(Vb,nB);
    else
        % Calculate positive & negative Riemann invariants
        rpos = (vel.*nB) + 2/(1.4-1)*cell_a;
        rneg = (free_vel.*nB) - 2/(1.4-1)*free_a;

        % Average invariants to get normal velocities
        un = (rpos+rneg)/2;

        % Obtain velocities at boundaries
        ub = vel + (un - vel).*nB;

        % Vb is primitive state vector at outflow boundary
        % Density is extrapolated
        % u,v are calculated
        % Pressure is extrapolated
        Vb = [V{i,j}(1) ub(1) ub(2) V{i,j}(4)];

        % Calculate East face flux
        e_flux{i,j} = flux(Vb,nB);
    end
% If east face is internal
else
    % Check to see is contravariant M >= 1
    if conM >= 1
        % Calculate east face flux using full flux
        e_flux{i,j} = flux(V{i,j},nE{i,j});
    else
        % Calculate east face flux using f+ and f-
        e_flux{i,j} = fpos(V{i,j},nE{i,j}) + fneg(V{i+1,j},nE{i,j});
    end
end


% Contravariant Mach for face
%conM = dot(vel,nN{i,j})/cell_a;
conM = (vel(1)*nN{i,j}(1) + vel(2)*nN{i,j}(2))/cell_a;

% If north face is invicid wall
if j == ncj
    % Boundary normal equal to face normal
    nB = nN{i,j};

    % Contravariant velocity:
    % Dot product of the u,v velocity of cell(i,ncj)
```

```
    % and the wall normal vector
    conV = dot(vel, nB);

    % Wall velocity:
    % Velocity of cell(i,ncj) - contravariant vel. * wall normal
    % vector
    velB = vel - conV*nB;

    % Vb is primitive state vector at the wall
    % Density is extrapolated
    % u,v are calculated
    % Pressure is extrapolated
    Vb = [V{i,j}(1) velB(1) velB(2) V{i,j}(4)];

    % Calculate north face flux
    n_flux{i,j} = flux(Vb,nB);
else
    % Check to see is contravariant M >= 1
    if conM >=1
        % Calculate north face flux using full flux
        n_flux{i,j} = flux(V{i,j},nN{i,j});
    else
        % Calculate north face flux using f+ and f-
        n_flux{i,j} = fpos(V{i,j},nN{i,j}) + fneg(V{i,j+1},nN{i,j});
    end
end


% Contravariant Mach for face
%conM = dot(vel,nW{i,j})/cell_a;
conM = (vel(1)*nW{i,j}(1) + vel(2)*nW{i,j}(2))/cell_a;

% If west face is inflow boundary
if i == 1
    % Boundary normal equal to face normal
    nB = nW{i,j};

    if conM >= 1
        % Assign boundary state vec to freestream vals
        Vb = V_free;

        % Calc flux for west face
        w_flux{i,j} = flux(Vb,nB);
    else
        % Calculate positive & negative Riemann invariant
        rpos = (vel.*nB) + 2/(1.4-1)*cell_a;
        rneg = (free_vel.*nB) - 2/(1.4-1)*free_a;

        % Average invariants to get normal velocities
        un = (rpos+rneg)/2;

        % Obtain velocities at boundaries
        ub = free_vel + (un + free_vel).*nB;

        % Vb is primitive state vector at inflow boundary
```

```
        % Density is freestream
        % u,v are calculated
        % Pressure is freesteam
        Vb = [V_free(1) ub(1) ub(2) V_free(4)];

        % Calculate west face flux
        w_flux{i,j} = flux(Vb,nB);
    end
else
    % Check to see is contravariant M >= 1
    if conM >=1
        % Calculate west face flux using full flux
        w_flux{i,j} = flux(V{i,j},nW{i,j});
    else
        % Calculate west face flux using the negative
        % of the adjacent cell east face flux
        w_flux{i,j} = -1*e_flux{i-1,j};
    end
end


% Contravariant Mach for face
%conM = dot(vel,nS{i,j})/cell_a;
conM = (vel(1)*nS{i,j}(1) + vel(2)*nS{i,j}(2))/cell_a;

% If south face is invicid wall
if j == 1
    % Boundary normal equal to face normal
    nB = nS{i,j};

    % Contravariant velocity:
    % Dot product of the u,v velocity of cell(i,1)
    % and the wall normal vector
    conV = dot(vel, nB);

    % Wall velocity:
    % Velocity of cell(i,1) - contravariant vel. * wall normal
    % vector
    velB = vel - conV*nB;

    % Vb is primitive state vector at the wall
    % Density is extrapolated
    % u,v are calculated
    % Pressure is extrapolated
    Vb = [V{i,j}(1) velB(1) velB(2) V{i,j}(4)];

    % Calculate south face flux
    s_flux{i,j} = flux(Vb,nB);
else
    % Check to see is contravariant M >= 1
    if conM >= 1
        % Calculate south face flux using full flux
        s_flux{i,j} = flux(V{i,j},nS{i,j});
    else
        % Calculate south face flux using the negative
```

```
                % of the adjacent cell north face flux
                s_flux{i,j} = -1*n_flux{i,j-1};
            end
        end

        % Assemble residual:
        % SUM(face flux * face area)
        resid{i,j} = e_flux{i,j}*sE(i,j) + ...
                     n_flux{i,j}*sN(i,j) + ...
                     w_flux{i,j}*sW(i,j) + ...
                     s_flux{i,j}*sS(i,j);
    end
end
```

flux.m

```
%% CALCULATE FULL FLUX VECTOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function f = flux(V,n)
% Inputs: Primitive state vector, V, from a cell
%         Normal vector, n, from a cell face

% Parse out variables and apply more names
rho = V(1);     % Density
u   = V(2);     % u velocity
v   = V(3);     % v velocity
P   = V(4);     % Pressure

%conV  = dot([u v],n);           % Contravariant velocity
conV  = u*n(1) + v*n(2);

% Assemble flux vector
f(1) = rho*conV;
f(2) = rho*u*conV + P*n(1);
f(3) = rho*v*conV + P*n(2);
f(4) = rho*h_0(V)*conV;
```

fpos.m

```
%% CALCULATE POSITIVE FLUX VECTOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function f = fpos(V,n)
% Inputs: Primitive state vector, V, from a cell
%         Normal vector, n, from a cell face

% Parse out variables and apply more names
rho = V(1);     % Density
u   = V(2);     % u velocity
v   = V(3);     % v velocity
P   = V(4);     % Pressure

gamma = 1.4;                    % Gamma
a     = speedsound(P,rho);      % Speed of sound
%conV  = dot([u v],n);           % Contravariant velocity
conV  = u*n(1) + v*n(2);
M     = conV/a;                 % Contravarient Mach number

% Assemble flux vector
f(1)  = rho*a/4*(M+1)^2;
f(2) = f(1) * (u + n(1)*(-conV+2*a)/gamma);
f(3) = f(1) * (v + n(2)*(-conV+2*a)/gamma);
f(4) = f(1) * (h_0(V) - a^2*(M-1)^2/(gamma + 1));
```

fneg.m

```
%% CALCULATE NEGATIVE FLUX VECTOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function f = fneg(V,n)
% Inputs: Primitive state vector, V, from a cell
%         Normal vector, n, from a cell face

% Parse out variables and apply more names
rho = V(1);     % Density
u   = V(2);     % u velocity
v   = V(3);     % v velocity
P   = V(4);     % Pressure

gamma = 1.4;                    % Gamma
a     = speedsound(P,rho);      % Speed of sound
%conV  = dot([u v],n);           % Contravariant velocity
conV  = u*n(1) + v*n(2);
M     = conV/a;                 % Contravarient Mach number

% Assemble flux vector
f(1)  = -rho*a/4*(M-1)^2;
f(2) = f(1) * (u + n(1)*(-conV-2*a)/gamma);
f(3) = f(1) * (v + n(2)*(-conV-2*a)/gamma);
f(4) = f(1) * (h_0(V) - a^2*(M+1)^2/(gamma + 1));
```

e_0.m

```
%% CALCULATE TOTAL ENERGY %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function e = e_0(V)
% Inputs: Primitive state vector from a cell

% Parse out variables and apply more names
rho = V(1);      % Density
u   = V(2);      % u velocity
v   = V(3);      % v velocity
P   = V(4);      % Pressure

% Total energy = pressure/[(gamma-1)*rho] + 1/2(u^2+v^2)
e = P/((1.4-1)*rho)+.5*(u^2+v^2);
```

h_0.m

```
%% CALCULATE TOTAL ENTHALPY %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function h = h_0(V)
% Inputs: Primitive state vector from a cell

% Parse out variables and apply more names
rho = V(1);      % Density
u   = V(2);      % u velocity
v   = V(3);      % v velocity
P   = V(4);      % Pressure

% Total enthalpy = total energy + pressure/rho
h = e_0(V)+P/rho;
```

convU_V.m

```
%% CALCULATE PRIMITIVE STATE VECTOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function V = convU_V(U)
% Inputs: Conservative state vector from a cell

% Parse out variables and apply more names
rho   = U(1);   % Density
rho_u = U(2);   % Density * u vel.
rho_v = U(3);   % Density * v vel.
rho_e = U(4);   % Density * total energy

% V = [rho u_vel v_vel pressure]
V(1) = rho;
V(2) = rho_u/rho;
V(3) = rho_v/rho;
V(4) = (rho_e - rho/2*(V(2)^2+V(3)^2))*(1.4-1);
```

convV_U.m

```
%% CALCULATE CONSERVATIVE STATE VECTOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function U = convV_U(V)
% Inputs: Primitive state vector from a cell

% Parse out variables and apply more names
rho = V(1);      % Density
u   = V(2);      % u velocity
v   = V(3);      % v velocity
P   = V(4);      % Pressure

% U = [rho rho*u_vel rho*v_vel rho*energy]
U(1) = rho;
U(2) = rho*u;
U(3) = rho*v;
U(4) = rho*e_0(V);
```

speedsound.m

```
%% CALCULATE SPEED OF SOUND %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function c = speedsound(P,rho)
% Inputs: Pressure and Density

% speed of sound = sqrt(gamma*pressure/rho)
c = sqrt(1.4*P/rho);
```

fixTime.m

```
%% CALCULATE TIME REMAINING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function time = fixTime(s)
% Input: time in seconds

min = fix(s/60);
sec = rem(s,60);
sec = fix(sec);

time = sprintf('%d:%02d',min,sec);
```