Analysis of One Dimensional Euler Equation Shocktube solution utilizing Parallel Computation

Christian Howard

May 8, 2013

Abstract

A formulation of the one dimensional Euler equations was constructed with application to the shocktube problem. The solution method utilized Finite Volume Methods with Roe's flux formulation for the fluxes and the Message Passing Interface API for parallel computation. An exact shocktube solver was constructed for use in the analysis aspect of the project, leading to insight on the convergence of the approximate solution. Errors were found to be largest at the location with the initial pressure discontinuity and at points on the exact solution where the slope was discontinuous. As the mesh was refined, the approximate solution curves converged to the exact solution.

Contents

1 Formulation

1.1 Conservative Form

The Euler equations in conservative form:

$$\frac{\partial \vec{Q}}{\partial t} + \frac{\partial \vec{F}}{\partial x} = \vec{0}$$

Also can be seen as the following:

$$\frac{\partial \vec{Q}}{\partial t} + \vec{\nabla} \cdot \left[\vec{F}, \vec{0}, \vec{0}\right] = \vec{0}$$

Where $\vec{Q} = [q_1, q_2, q_3]^T = [\rho, \rho u, \rho E]^T$ and $\vec{F} = [\rho u, (\rho u^2 + p), (\rho E + p)u]^T$

1.2 Primitive Form

The primitive form, also known as the quasi-linear form, is useful since it is the form used to get to the characteristic form:

$$\frac{\partial \vec{V}}{\partial t} + A\frac{\partial \vec{V}}{\partial x} = \vec{0}$$

Where $\vec{V} = [v_1,\ v_2,\ v_3]^T = [\rho,\ u,\ p]^T$ and $\mathrm{A} = \begin{bmatrix} u & \rho & 0 \\ 0 & u & \frac{1}{\rho} \\ 0 & p\gamma & u \end{bmatrix}$

1.3 Characteristic Form

The characteristic form is useful for applying the boundary conditions:

$$\frac{\partial \vec{W}}{\partial t} + \Lambda\frac{\partial \vec{W}}{\partial x} = \vec{0}$$

Where $\vec{W} = [w_1,\ w_2,\ w_3]^T$, $\Lambda = \begin{bmatrix} u & 0 & 0 \\ 0 & u-c & 0 \\ 0 & 0 & u+c \end{bmatrix}$

and $c = \sqrt{\frac{p\gamma}{\rho}}$ is the speed of sound.

1.4 Transformation Relationships

1.4.1 $\vec{Q} \Leftrightarrow \vec{V}$

The relationship is that $\partial\vec{Q} = \frac{\partial\vec{Q}}{\partial\vec{V}}\partial\vec{V}$, or $\partial\vec{V} = \frac{\partial\vec{V}}{\partial\vec{Q}}\partial\vec{Q}$

$$Where\ \frac{\partial\vec{Q}}{\partial\vec{V}} = \begin{bmatrix} 1 & 0 & 0 \\ u & \rho & 0 \\ \frac{u^2}{2} & \rho u & (\gamma-1)^{-1} \end{bmatrix}\ and\ \frac{\partial\vec{V}}{\partial\vec{Q}} = \begin{bmatrix} 1 & -\frac{u}{\rho} & \frac{u^2(\gamma-1)}{2} \\ 0 & \frac{1}{\rho} & -(\gamma-1)u \\ 0 & 0 & (\gamma-1) \end{bmatrix}$$

1.4.2 $\vec{V} \Leftrightarrow \vec{W}$

The relationship is that $\partial\vec{V} = \frac{\partial\vec{V}}{\partial\vec{W}}\partial\vec{W}$, or $\partial\vec{W} = \frac{\partial\vec{W}}{\partial\vec{V}}\partial\vec{V}$

$$Where\ \frac{\partial\vec{V}}{\partial\vec{W}} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3] = X = \begin{bmatrix} 1 & \frac{1}{c^2} & \frac{1}{c^2} \\ 0 & -\frac{1}{\rho c} & \frac{1}{\rho c} \\ 0 & 1 & 1 \end{bmatrix}$$

$$and\ \frac{\partial\vec{W}}{\partial\vec{V}} = X^{-1} = \begin{bmatrix} 1 & 0 & -\frac{1}{c^2} \\ 0 & -\frac{\rho c}{2} & \frac{1}{2} \\ 0 & \frac{\rho c}{2} & \frac{1}{2} \end{bmatrix}$$

1.4.3 $A \Leftrightarrow \Lambda$

The relationships are the following:

$$\Lambda = X^{-1}AX$$
$$\mathrm{A} = X\Lambda X^{-1}$$

1.5 Discretization

1.5.1 Governing Equation Discretization

The problem statement for this formulation is the following:

Given that the domain $\Omega =]0,l[$ and given the time $\mathrm{T} =]0,\tau[$, we want to find a solution for $\vec{Q} \in \Omega \times T \ni$ the following equations are satisfied:

$$\frac{\partial \vec{Q}}{\partial t} + \vec{\nabla} \cdot \left[\vec{F}, \vec{0}, \vec{0}\right] = \vec{0}$$

Now, we wish for these equations to hold over Ω, so we integrate the above equation across Ω to obtain the simplified form below:

$$\int_{\Omega} \left(\frac{\partial \vec{Q}}{\partial t} + \vec{\nabla} \cdot \left[\vec{F}, \vec{0}, \vec{0}\right] \right) d\Omega = \vec{0} \tag{1}$$

$$\int_{\Omega} \frac{\partial \vec{Q}}{\partial t} d\Omega + \int_{\Gamma} \left[\vec{F}, \vec{0}, \vec{0}\right] \cdot \vec{n} d\Gamma = \vec{0} \tag{2}$$

$$\frac{\partial}{\partial t} \int_{\Omega} \vec{Q} d\Omega + \int_{\Gamma} \vec{F} n_x \, d\Gamma = \vec{0} \tag{3}$$

Now, to discretize the equations, the domain is broken up into N cells and the desire is that each element carries an average value for the solution across its local domain, denoted as $\vec{\bar{Q}}$. It is also expected that each individual element will satisfy local conservation, making this method globally conservative. Thus, the following is the discretization:

$$\sum_{i=1}^{N} \left[\frac{\partial}{\partial t} \int_{\Omega_i} \vec{Q} d\Omega_i + \int_{\Gamma_i} \vec{F} n_x d\Gamma_i \right] = \vec{0} \tag{4}$$

$$\sum_{i=1}^{N} \left[\Delta x_i \frac{\partial \vec{\bar{Q}}_i}{\partial t} + \vec{F}(x_{i+\frac{1}{2}}) - \vec{F}(x_{i-\frac{1}{2}}) \right] = \vec{0} \tag{5}$$

$$\frac{\partial \vec{\bar{Q}}_i}{\partial t} + \frac{\vec{F}(x_{i+\frac{1}{2}}) - \vec{F}(x_{i-\frac{1}{2}})}{\Delta x_i} = \vec{0} \tag{6}$$

1.5.2 Roe Scheme Fluxes

For the fluxes,$\vec{F}(x_{i+\frac{1}{2}})$, the solution for this project utilized Roe's Scheme, with information ont it found in [1]. Philip L. Roe developed this scheme in 1981 with a goal in mind to improve on Godunov's method. The idea was to take the quasi-linear form

$$\frac{\partial \vec{V}}{\partial t} + A(\vec{V})\frac{\partial \vec{V}}{\partial x} = \vec{0}$$

And obtain an $\hat{A}(\vec{V}_L, \vec{V}_R) = A(\vec{V})$ given

$$\vec{V}(x,0) = \begin{cases} \vec{V}_L, \ x < 0 \\ \vec{V}_R, \ x > 0 \end{cases}$$

with the following properties:

1. The equation should maintain the hyperbolicity. This implies that $\hat{A}$ should have purely real eigenvalues and corresponding eigenvectors
2. Jacobian Consistency:

$$\hat{A}(\vec{V}, \vec{V}) = A(\vec{V})$$

3. Conservation:

$$\vec{F}(\vec{V}_R) - \vec{F}(\vec{V}_L) = \hat{A}(\vec{V}_L, \vec{V}_R)(\vec{V}_R - \vec{V}_L)$$

Roe found that for the Euler equations, $\hat{A} = A$ when you evaluate the quantities at what he called the Roe average, which had the form:

$$\hat{\phi} = \frac{\sqrt{\rho_L}\phi_L + \sqrt{\rho_R}\phi_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}$$

Where ϕ is a quantity you are finding the Roe average for. An example can be finding the velocity Roe average, like below:

$$\hat{u} = \frac{\sqrt{\rho_L}u_L + \sqrt{\rho_R}u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}$$

At this point, the goal is to obtain $F_{i+\frac{1}{2}}(q_L, q_R)$. Using the method of characteristics, one can break down the left and right states of the solution into their characteristic amplitudes, which have the following form:

$$\vec{V}_L = \sum_{r=1}^{3} \eta_p \hat{\mathbf{x}}_p \tag{7}$$

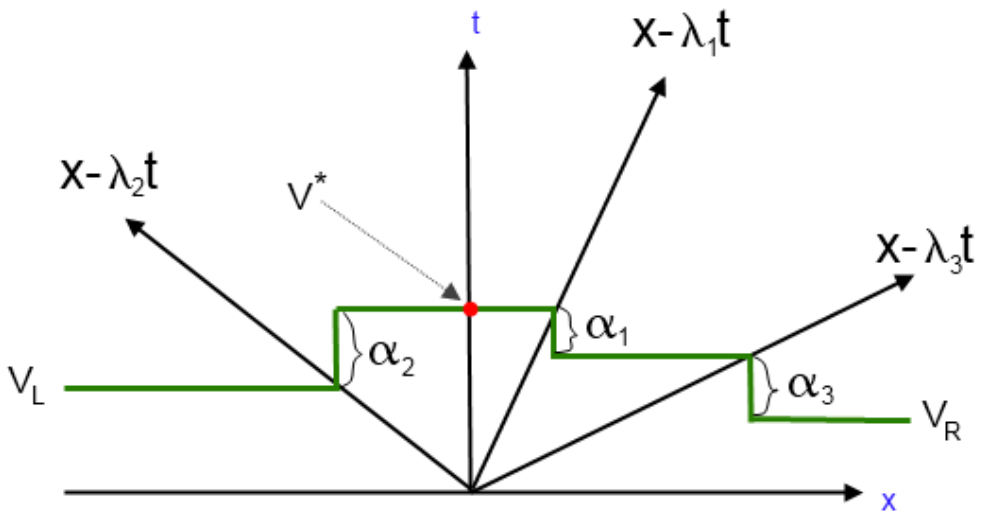
$$\vec{V}_R = \sum_{r=1}^{3} \zeta_p \hat{\mathbf{x}}_p \tag{8}$$

where η_p and ζ_p are the characteristic amplitudes for the left and right states and $\hat{\mathbf{x}}_p$ are the eigenvectors of A evaluated at the Roe averages.

Subtracting equation (8) from (7), the following is obtained:

$$\vec{V}_R - \vec{V}_L = \sum_{r=1}^{3} \alpha_p \hat{\mathbf{x}}_p$$

where α_p are the jumps of each characteristic variable.



x-t plane with Characteristic curves at flux interface

Solving for the value of the solution along the characteristic ray x/t = 0, we obtain

$$\vec{V}^* = \vec{V}_L + \sum_{\lambda_p<0} \alpha_p \hat{\mathbf{x}}_p \tag{9}$$

$$= \vec{V}_R - \sum_{\lambda>0} \alpha_p \hat{\mathbf{x}}_p \tag{10}$$

These two separate equations can be combined to generate the following relationship:

$$\vec{V}^* = \frac{\vec{V}_L + \vec{V}_R}{2} + \frac{1}{2}\left\{ \sum_{\lambda_p<0} \alpha_p \hat{\mathbf{x}}_p - \sum_{\lambda_p>0} \alpha_p \hat{\mathbf{x}}_p \right\} \tag{11}$$

After this step, the numerical flux, $F_{i+\frac{1}{2}}$ is found to be:

$$F_{i+\frac{1}{2}} = \hat{A}\vec{V}^* = \frac{1}{2}\left[\hat{A}(\vec{V}_L + \vec{V}_R) - \sum_{p=1}^{3} |\lambda_p| \alpha_p \hat{\mathbf{x}}_p \right] \tag{12}$$

$$= \frac{1}{2}\left[(f(\vec{V}_L) + f(\vec{V}_R)) - \sum_{p=1}^{3} |\lambda_p| \alpha_p \hat{\mathbf{x}}_p \right] \tag{13}$$

The expressions above are the resulting Roe scheme fluxes. These fluxes can be observed to be a simple average of the exact fluxes using both sets of data on a boundary minus a summation term. Plugging in these flux quantities into equation (6), one will find the expression equates to a central finite difference with just enough dissipation to keep the solution stable.

One last point to make is that due to the linearized form of this solution scheme, experiments with it in the past have shown that an entropy spike occures when $|\hat{u}| \approx \hat{c}$. The fix, which is for eigenvalues λ_2 and λ_3, is the following:

$$For\ |\lambda_p| < \epsilon, where\ \epsilon << 1$$
$$Then\ \lambda_p = \frac{1}{2}\left(\frac{\lambda_p^2}{\epsilon} + \epsilon\right)$$

As a note, the choice of ϵ is arbitrary as long as it is small compared to unity, as shown in the statement above. An ϵ value of $1 \cdot 10^{-10}$ was used for the analysis later on in this document.

To finalize the Roe Method, the following quantities are used in the above formulation:

$$\lambda_1 = \hat{u},\ \hat{\mathbf{x}}_1 = [1, \hat{u}, \hat{u}^2/2]^T \tag{14}$$
$$\lambda_2 = \hat{u} - \hat{c},\ \hat{\mathbf{x}}_2 = \left[1, \hat{u} - \hat{c}, \hat{h} - \hat{u}\hat{c}\right]^T \tag{15}$$
$$\lambda_3 = \hat{u} + \hat{c},\ \hat{\mathbf{x}}_3 = \left[1, \hat{u} + \hat{c}, \hat{h} + \hat{u}\hat{c}\right]^T \tag{16}$$
$$\hat{h} = \frac{\hat{c}^2}{(\gamma - 1)} + \frac{\hat{u}^2}{2} \tag{17}$$

And the jump values can be found using the characteristics to be the following quantities:

$$\alpha_1 = \Delta w_1 = X^{-1}\Delta\vec{V} \cdot \hat{e}_1 = \Delta\rho - \frac{\Delta p}{\hat{c}^2} \tag{18}$$
$$\alpha_2 = \Delta w_2 = X^{-1}\Delta\vec{V} \cdot \hat{e}_2 = \frac{1}{2\hat{c}^2}(\Delta p - \hat{c}\hat{\rho}\Delta u) \tag{19}$$
$$\alpha_3 = \Delta w_3 = X^{-1}\Delta\vec{V} \cdot \hat{e}_3 = \frac{1}{2\hat{c}^2}(\Delta p + \hat{c}\hat{\rho}\Delta u) \tag{20}$$

Where $\hat{\rho} = \sqrt{\rho_L \rho_R}$ and for any quantity ϕ, the following holds:

$$\Delta\phi = \phi_R - \phi_L$$

Note that this difference is not a difference of Roe averaged values for the quantity ϕ.

1.5.3 Boundary Conditions

For the shocktube problem being examined, I am building in solid wall boundary conditions on each end. This implies the velocity at the wall point will be zero at all times, so thus the differential expression $du = 0$ is true. To relate this to some meaningful quantities in the formulation, a shift to looking at the characteristic variables is needed. It is understood that $d\vec{V} = Xd\vec{W}$. This relationship expanded gives the following results:

$$\begin{bmatrix} d\rho \\ du \\ dp \end{bmatrix} = \begin{bmatrix} dw_1 + \frac{dw_2+dw_3}{c^2} \\ \frac{dw_3-dw_2}{\rho c} \\ dw_2 + dw_3 \end{bmatrix}$$

Based off of this, we come to find an expression for du. Setting $du = 0$, we obtain the following:

$$dw_3 - dw_2 = 0$$

This is the characteristic relationship we will now utilize. Since the incoming characteristic on the left boundary is w_3, this means we can change this value but not w_2's value. So on the left boundary, we will set $\frac{\partial w_3}{\partial t} = \frac{\partial w_2}{\partial t}$. The converse is true on the right boundary, so $\frac{\partial w_2}{\partial t} = \frac{\partial w_3}{\partial t}$ is set there.

The process utilized in the end for solving the boundary condition is the following:

1. Obtain approximate derivative values for $\frac{\partial \vec{Q}}{\partial t}$ on each boundary
2. Using these values, obtain $\frac{\partial \vec{V}}{\partial t}$ by using $\frac{\partial \vec{V}}{\partial \vec{Q}}$
3. Using these values, obtain $\frac{\partial \vec{W}}{\partial t}$ by using $\frac{\partial \vec{W}}{\partial \vec{V}}$
4. Change the $\frac{\partial \vec{W}}{\partial t}$ vector according to the boundary being looked at
5. Rotate $\frac{\partial \vec{W}}{\partial t}$ back into $\frac{\partial \vec{V}}{\partial t}$ and then rotate $\frac{\partial \vec{V}}{\partial t}$ back into $\frac{\partial \vec{Q}}{\partial t}$ and insert into output derivative vector

1.5.4 General Solution Process

1. Create initial condition vector and spacial grid
2. Specify time frame to solve the problem
3. Plug in the above quantities into an adaptive Runge-Kutta integrator
   (a) Obtain $\frac{\partial \vec{Q}}{\partial t}$ for time t_k
      i. Obtain the Fluxes at each interface
      ii. Use fluxes to find $\frac{\partial \vec{Q}}{\partial t}$ for each cell but the ones on the global boundary
      iii. Apply boundary conditions to obtain $\frac{\partial \vec{Q}}{\partial t}$ for the cells on the global boundary
      iv. Return the resulting $\frac{\partial \vec{Q}}{\partial t}$ vector
   (b) Use the various $\frac{\partial \vec{Q}}{\partial t}$ vectors to obtain an error estimate and adaptively change step size if needed
   (c) Obtain new solution for t = t_{k+1}
   (d) Output the percentage of the solution Completed
   (e) Write the solution to file
   (f) Go back to (a) until t_{k+1} equals the end time
4. Stitch file data together if using multiple threads and do any necessary post processing

2 Parallelization of Solution Method

2.1 Introduction

To make it possible to obtain a solution with higher accuracy and efficiency, the finite volume solution method utilized was made to be parallel. Due to the fact that one process would need to obtain data from neighboring processes, the Message Passing Interface API was utilized to do the parallel computation, explained in [2]. This was chosen due to the capabilities to send data between processes easily and efficiently. Now, the figure below shows an example of how the domain was broken up so that each process had a chunk to work with.

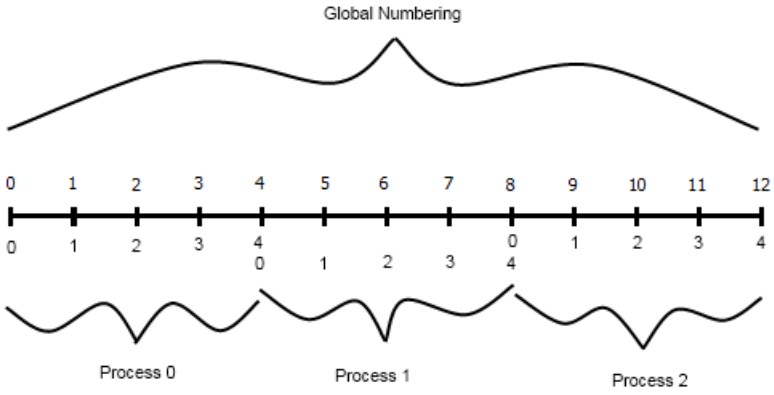


Figure P1: Domain Separation into Processes

Since the Roe scheme was utilized for the interior data, this required using the left and right neighbor of a cell to obtain the fluxes which were used to find how much the average value of the solution on the cell will change. As one might see in the figure, cells on the edge of their domain chunk aren't able to access one of their neighbors directly, since one of their neighbors is in a different thread.

To make the solution method parallel, aside from sending only chunks of the domain to each thread, the fluxes had to be obtained by swapping data between processes. After the fluxes were obtained, however, the rest of the original finite volume method remained the same for the given domain piece a process had. Below is a description of the special measures taken with the flux obtaining process.

2.2 Flux Portion Parallelization

2.2.1 Parallel Variable Definitions

N_n: Number of cells per process

N_p: Number of processes

N_f: Number of fluxes per process

N_{ID}: ID value for a process, going from 0,1,2 ... $(N_p - 1)$

2.2.2 Preliminary Remarks

1. In any given process with N_n cells, the number of fluxes it had to store is given by the boolean relationship

$$N_f = (N_n - 1) + (N_{ID} < (N_p - 1)) + (N_{ID} > 0)$$

   (a) The reasoning for this is that the boundary processes will need one more flux than the number of interior fluxes and the interior processes will need two more fluxes than their interior number

   (b) Note the interior number of fluxes for any process is $N_n - 1$

2.2.3 Implementation

The steps:

1. Obtain all the interior flux values for each process

2. If $N_{ID} < N_p - 1$ for a process, send the flux with local indice $N_n - 2$ in a process with N_{ID} to the process with $N_{ID} + 1$ and into the local flux indice 0 in this new thread

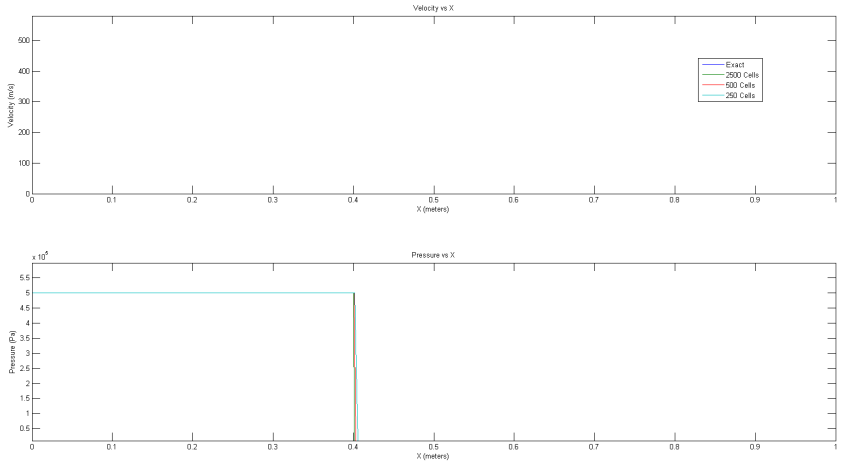
3. If $N_{ID} > 0$ for a process, send the flux in local indice 1 for a thread with N_{ID} to a process with $N_{ID} - 1$ and into the local flux indice $N_n - 1$ in this new thread

4. Return the resulting flux vector and use to find the time derivatives of each cell within the process

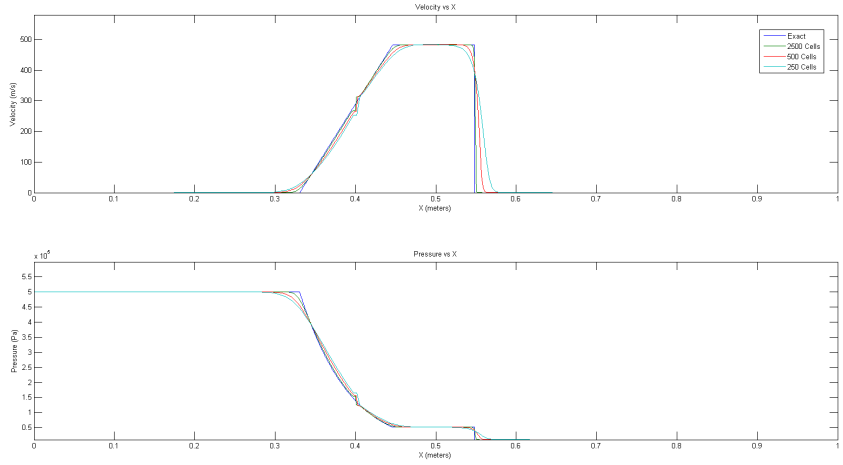
3 Analysis of Results

3.1 Results

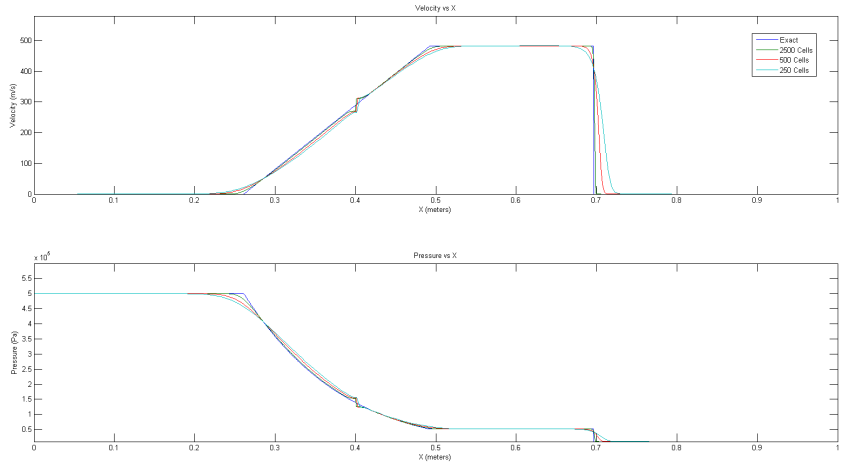
Within this section, an analysis has been done to compare the velocity and pressure exact solution with the numerical one.The numerical solutions utilized 250, 500 and 2500 cells to approximate the solution. Shown below are a series of figures of these numerical solutions and the exact solution at times t = 0.0, $2{\cdot}10^{-4}$,$4{\cdot}10^{-4}$,$6{\cdot}10^{-4}$, and $8{\cdot}10^{-4}$ seconds.



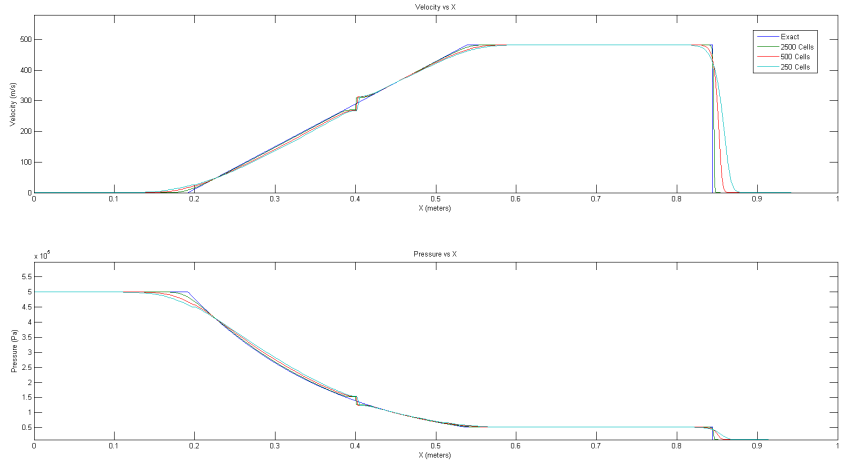
Velocity and Pressure plots
at t = 0.0 seconds



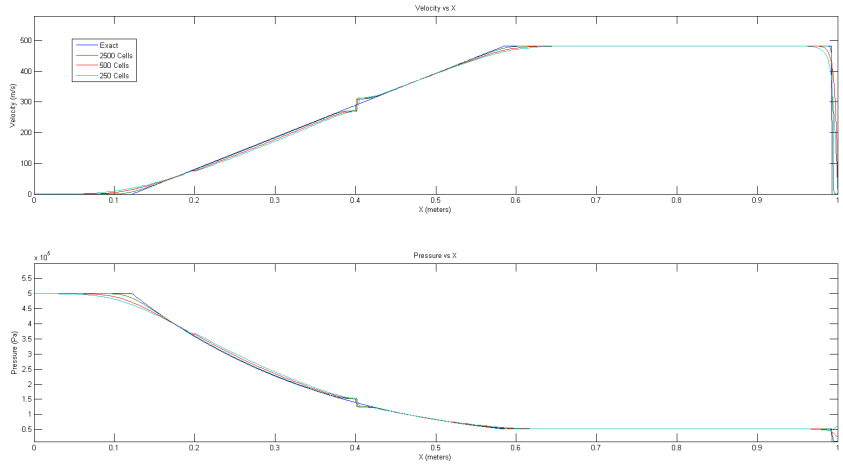
Velocity and Pressure plots
at t = $2{\cdot}10^{-4}$ seconds



Velocity and Pressure plots
at t = $4{\cdot}10^{-4}$ seconds



Velocity and Pressure plots
at t = $6{\cdot}10^{-4}$ seconds



Velocity and Pressure plots
at t = $8{\cdot}10^{-4}$ seconds

3.2 Discussion

Just looking through these figures, it is simple to see a difference in the exact solution and the approximate ones. At the location of the initial pressure discontinuity, the approximate solutions carry this discontinuity throughout all time, while the exact solution has it disappear due to the expansion fan. One positive note to make,however, is that the size of this discontinuity does appear to shrink as the mesh is refined.

The approximate solutions also have a smooth curve at any location where the exact solution has a discontinuous change in slope. These regions make it easy to see the convergence of the solution, as the approximate solution more tightly fits to the exact solution curve as the mesh is refined.

Keeping these observed flaws in the approximate solution scheme in mind, the approximate solutions fit to the exact solution quite well in the other regions. Some things to take away from the analysis is that if one wants to utilize this solution scheme and desires very accurate results, striving to utilize a parallel approach to the computation is ideal.

This idealization is due to the need for not only greater computational speed, but greater computational resources, so more cells can be used in the approximation, that could be obtained by using a cluster or something along those lines. The other possibility is to obtain a higher order scheme with *just enough* dissipation, just as Roe's Method has, and push to parallel this scheme to obtain greater accuracy.

4 Conclusion

The Euler Equations, applied to a shocktube problem, were solved utilizing the Finite Volume Method with fluxes found using Roe's Method and a parallel algorithm for efficiency and added resources amongst multiple processors. At approximations using high numbers of cells, the approximate solution using the methods specified converged towards the exact solution. This was seen by having a few numerical solutions using different numbers of cells and seeing the trend that increased number of cells brought the solution curve closer to being the exact solution. However, there were still noticeable errors at the membrane location due to the initial, large discontinuity in pressure. There were also found to be errors at places where the slope was discontinuous, though they grew considerably smaller as the number of cells increased. Future goals are to find higher order Finite Volume methods, whether through polynomial reconstruction schemes within cells or by other means, and continue to exploit the benefits of parallelism.

5 Appendix

5.1 Code

5.1.1 Global Variables

This section is very brief, but in my implementation I had two global variables that had the following functions in the grand scheme of things:

1. One variable, denoted by *myid* in the code below, is a variable initialized at the start that holds the ID for a given process. This is important in the code to know whether a process is considered a boundary process or an interior one, and helps know which other threads it will need to swap data with.

2. The other variable, denoted by *numprocs*, is a variable initialized to know the number of processes being used. This is important to know which processes should be interior and boundary processes.

5.1.2 Vector Structure

```
/*
Vector structure
-Holds a dynamic array of cnt elements of type double
*/
typedef struct
{
        double* v;
        int cnt;

}Vector;
```

```
/*Initialization of a dynamic vector of length num of
zero element values*/
Vector* v_init(int num)
{
        int i = 0;
        Vector* o = (Vector*)malloc(sizeof(Vector));
        o->v = malloc(sizeof(double)*num);
        for(i = 0; i<num; i++)
                o->v[i] = 0;
        o->cnt = num;
        return o;
}

/*Initialization of a dynamic vector with elements
comprised of the list specified*/
Vector* v_initlist(int num,...)
{
        int i=0;
        Vector* o = v_init(num);
        va_list list;
        va_start(list,num);
        for(i = 0; i<num; i++)
        {
                o->v[i] = va_arg(list,double);
        }
        o->cnt = num;

        va_end(list);
        return o;
}

/*Free memory associated with a dynamic Vector*/
void v_free(Vector* v)
{
        free(v->v);
        free(v);
}
```

```
/* Print a vector's contents to the terminal */
void v_print(Vector* v)
{
        int i;
        if(v == NULL)
                return;
        if(v->cnt == 0)
                return;
        printf("< ");
        for(i = 0; i<v->cnt; i++)
        {       if(i == 0)
                        printf("%lf",v->v[i]);
                else
                        printf(", %lf",v->v[i]);
        }
        printf(" >\n");
}

/*Create a copy of the vector */
Vector* v_copy(Vector* v)
{
        Vector* o = v_init(v->cnt);
        int i = 0;
        for(i = 0; i<v->cnt; i++)
                o->v[i] = v->v[i];
        return o;
}
```

```
/*Initialization of a dynamic vector by creating
a linear space between a start and end value
with numnode number of nodes in the space*/
Vector* v_linspace(double start,
                   double end,
                   int numnode)
{
        Vector* o = v_init(numnode);
        double dx = (end-start)/(double)(numnode-1);
        int i = 0;

        for( i = 0; i < numnode; i++)
                o->v[i] = start+i*dx;
        return o;
}
```

5.1.3 MPI Helper Functions

```
/*Initialize MPI and obtain the number
of processes being used and the ID for
each of the used processes*/
void MPI_Euler_Init(int argc, char * argv[])
{
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
}


/*Have a process print its ID*/
void printID()
{
        printf("This id is %i\n",myid);
}
```

```
/*Create a linear space of values across
all the processes*/
Vector* mpi_linspace(double start, //Start value
                     double end,   //End value
                     int num)      //Number of nodes desired
{
    double dL = end-start;
    Vector* out = v_linspace(start+(myid/(double)numprocs)*dL,
                  start+((myid+1)/(double)numprocs)*dL,
                  num);
        return out;
}

/*Create the shocktube initial condition
across all the processes based on their
portion of the domain*/
Vector* mpi_Shocktube_IC( Vector* nodes, //Spacial grid for process
                          double P_high, //High pressure region value
                          double P_low,  //Low pressure region value
                          double Temperature, //Temperature of gas
                          double SpecHeatRatio, //Specific heat ratio
                          double memloc ) //Location of membrane
{
        return e_ShocktubeIC(P_high,
                             P_low,
                             Temperature,
                             SpecHeatRatio,
                             memloc,
                             nodes);
}
```

```
/*Take an initial file name and
create multiple files accordingly
for the number of processes being
used in the problem*/
char* mpi_newFileName( char* name )
{
        int cnt = 0,i;
        while( name[cnt] != '\0' )
                cnt++;
        char c = '0' + myid;
        int size = cnt +
                   4 +      // 4 for '.txt' part,
                   1 +      // 1 for the c part
                   1;       // and 1 for /0

        //Allocate memory for a new file name
        char* nname = malloc( sizeof(char) * (cnt+6) );

        //Create the ending to the new name
        nname[size-1] = '\0';
        nname[cnt] = c;
        nname[cnt+1] = '.';
        nname[cnt+2] = 't';
        nname[cnt+3] = 'x';
        nname[cnt+4] = 't';

        //Fill in the beginning of the
        //new name with the original one
        for( i = 0; i < cnt; i++ )
                nname[i] = name[i];

        return nname;
}
```

5.1.4 Finite Volume Functions

```
/*
Function for creating local Shocktube
Initial conditions
*/
Vector* e_ShocktubeIC(double highpressure, //High Pressure value
                      double lowpressure,  //Low pressure value
                      double temp,         //Temperature of gas
                      double gamma,        //Specific heat ratio
                      double memloc,       //Membrane location
                      Vector* nodepos)     //Spacial vector
{
        //Create initial condition vector
        Vector* IC = v_init(3*nodepos->cnt);

        //Construct and initialize
        //iteration variable
        int i = 0;
        for(i = 0; i<nodepos->cnt; i++)
        {
                if(nodepos->v[i]<=memloc)
                { //If values left of the membrane

                IC->v[i+2*nodepos->cnt] = highpressure/(gamma-1);
                IC->v[i] = highpressure/(287*temp);

                }else
                {//If values right of the membrane

                IC->v[i+2*nodepos->cnt] = lowpressure/(gamma-1);
                IC->v[i] = lowpressure/(287*temp);

                }
        }
        //Return resulting initial condition
        return IC;
}
```

```
/*
Function for doing the Roe Scheme
entropy fix
*/
double e_roefix(double lambda){

        if( fabs(lambda) < DELTA )
                lambda = .5*( (lambda*lambda/DELTA) + DELTA );

        return lambda;

}


/*Function for constructing and obtaining
the Roe fluxes at the cells' interfaces*/
void e_1Dflux(Vector* data,
                              double gamma,
                              Vector* output)
{
        //Left and right values of
        //primitive variables
        double rL = 0, uL = 0, pL = 0;
        double rR = 0, uR = 0, pR = 0;

        //Left and right values of
        //conservative variables
        double q1L, q2L, q3L, q1R, q2R, q3R;

        //Change in primitive variables
        double dr = 0, du = 0, dp = 0;

        //Exact flux variables on both
        //sides of flux interface
        double frhoL = 0, frhoR = 0, fuL = 0;
        double fuR = 0, fpL = 0, fpR = 0;

        //Roe Scheme variables
```

```
double ravg = 0, wrL = 0, wrR = 0, wsum = 0;
double sumr = 0, sumu = 0, sump = 0;
double RHO, U, P,H,C;
double jump, lambda = 0, temp = 0;

//Initialize an MPI Status var
MPI_Status status;

//Variables for iteration purposes
int i = 0,tmp = 0;
int numnodes = (data->cnt/3);
int numelem = numnodes-1;
int chck1 = myid > 0;
int chck2 = myid < numprocs-1;

//Obtain number of overall fluxes
//for the current process
tmp = (numelem+chck1+chck2);

//Start loop over flux positions
for( i = 1; i<=numelem; i++)
{

//Obtain conservative variables on each side
//of the flux interface
q1L = data->v[i-1];
q2L = data->v[(i-1)+numnodes];
q3L = data->v[(i-1)+2*numnodes];

q1R = data->v[i];
q2R = data->v[i+numnodes];
q3R = data->v[i+2*numnodes];

//Obtain the left cell variable values
rL = q1L;
uL = q2L/rL;
pL = (gamma-1)*(q3L - .5*q2L*q2L/q1L);
```

```
//Obtain the right cell variable values
rR = q1R;
uR = q2R/rR;
pR = (gamma-1)*(q3R - .5*q2R*q2R/q1R);

//Find the difference of dependent
//variable values across cells
dr = rR-rL; du = uR-uL; dp = pR-pL;

//Find the weighting values based on rho
wrL = sqrt(rL); wrR = sqrt(rR);
wsum = wrL + wrR;

/*-----------------------------------------

<< Obtain Roe Averaged Quantities >>

-----------------------------------------*/

//Obtain the Roe Averaged Density
RHO = wrL*rL+wrR*rR;
RHO /= wsum;

//Obtain the Roe Averaged Density
U = wrL*uL + wrR*uR;
U /= wsum;

//Obtain the Roe Averaged Density
P = wrL*pL + wrR*pR;
P /= wsum;

//Obtain the geometric average of density
ravg = wrL*wrR;

//Obtain the Roe Averaged speed of sound
C = sqrt(gamma*P/RHO);

//Obtain the Roe averaged enthalpy
```

```
H = C*C/(gamma-1) + U*U/2.0;

//Find flux portion from first jump
jump = dr - dp/(C*C);
lambda = U;

sumr = fabs(lambda)*jump;
sumu = sumr*U;
sump = sumu*U*.5;

//Find flux portion from second jump
jump = (dp - C*ravg*du)/(2*C*C);

lambda = e_roefix(U-C);
temp = fabs(lambda)*jump;

sumr = sumr + temp;
sumu = sumu + temp*(lambda);
sump = sump + temp*(H-U*C);

//Find flux portion from third jump
jump = (dp + C*ravg*du)/(2*C*C);

lambda = e_roefix(U+C);
temp = fabs(lambda)*jump;

sumr = sumr + temp;
sumu = sumu + temp*(lambda);
sump = sump + temp*(H+U*C);

//Calculate the exact fluxes
frhoL = q2L;
frhoR = q2R;
fuL = rL*uL*uL + pL;
fuR = rR*uR*uR + pR;
fpL = (q3L+pL)*uL;
fpR = (q3R+pR)*uR;
```

```
/*----------------------------------------

     << Calculate Roe Fluxes >>

----------------------------------------*/
/*<< Flux for the density variable >>*/
output->v[i-1+chck1] = .5*(frhoL+frhoR - sumr);

/*<< Flux for the velocity variable >>*/
output->v[(i-1+chck1)+tmp] = .5*(fuL+fuR-sumu);

/*<< Flux for the pressure variable >>*/
output->v[(i-1+chck1)+2*tmp] = .5*(fpL + fpR - sump);
}

if( chck2 ) //If process isn't the right boundary one
{
        //Send data to next process
        MPI_Send(&output->v[tmp-2],
                        1, MPI_DOUBLE_PRECISION,
                            myid+1, 1 , MPI_COMM_WORLD);
        MPI_Send(&output->v[2*tmp-2],
                        1,MPI_DOUBLE_PRECISION,
                            myid+1, 2 , MPI_COMM_WORLD);
        MPI_Send(&output->v[3*tmp-2],
                        1,MPI_DOUBLE_PRECISION,
                            myid+1, 3 , MPI_COMM_WORLD);
}
if( chck1 ) //If the process isn't the left boundary
{
        //Grab data from previous process
        MPI_Recv(&output->v[0],
                        1,MPI_DOUBLE_PRECISION,
                            myid-1, 1 , MPI_COMM_WORLD,&status);
        MPI_Recv(&output->v[tmp],
                        1,MPI_DOUBLE_PRECISION,
                            myid-1, 2 , MPI_COMM_WORLD,&status);
        MPI_Recv(&output->v[2*tmp],
```

```
                                1,MPI_DOUBLE_PRECISION,
                                    myid-1, 3 , MPI_COMM_WORLD,&status);
        }

        if( chck1 ) //If the process isn't the left boundary
        {
                //Send data to previous process
                MPI_Send(&output->v[1],
                                1,MPI_DOUBLE_PRECISION,
                                    myid-1, 1 , MPI_COMM_WORLD);
                MPI_Send(&output->v[1+tmp],
                                1,MPI_DOUBLE_PRECISION,
                                    myid-1, 2 , MPI_COMM_WORLD);
                MPI_Send(&output->v[1+2*tmp],
                                1,MPI_DOUBLE_PRECISION,
                                    myid-1, 3 , MPI_COMM_WORLD);
        }
        if( chck2 ) //If process isn't the right boundary one
        {
                //Grab data from next process
                MPI_Recv(&output->v[tmp-1],
                                1, MPI_DOUBLE_PRECISION,
                                    myid+1, 1 , MPI_COMM_WORLD,&status);
                MPI_Recv(&output->v[2*tmp-1],
                                1,MPI_DOUBLE_PRECISION,
                                    myid+1, 2 , MPI_COMM_WORLD,&status);
                MPI_Recv(&output->v[3*tmp-1],
                                1,MPI_DOUBLE_PRECISION,
                                    myid+1, 3 , MPI_COMM_WORLD,&status);
        }
}


/* The Finite Volume solution for the Shocktube
Problem, based on the Euler Equations*/
void e_Shocktube(Vector* q,
                 Vector* grid,
                 Vector* qdot)
```

```
{

        //Number of nodes in grid
        int len = grid->cnt;

        //Number of fluxes for the partition
        //of the domain
        int fluxcount = len-1 +
                        (myid > 0) +
                        (myid < numprocs-1);

        //A check that the partition is
        //not the left boundary
        int chck1 = myid > 0;

        //Find the length of a cell
        double dx = grid->v[1]-grid->v[0];

        //Initialize overall flux vector
        Vector* flux = v_init(3 * fluxcount );

        //Initialize local flux vectors
        Vector *fluxL = v_init(3), *fluxR = v_init(3),*tmp = NULL;

        //Create variables for use in
        //later computations
        int i = 0;

        double rho, u, p, c,c2;
        double tmp1 = 0, tmp2 = 0;
        double gamma = 1.4;
        double q11, q21, q31, q12,
        double q22, q32, q13, q23, q33;

        double drhodt, dudt, dpdt;
        double dq1dt, dq2dt, dq3dt;
        double dq1dx,dq2dx,dq3dx;
        double dw1dt, dw2dt, dw3dt;
```

```
        //Obtain the fluxes
        e_1Dflux(q, gamma, flux);

        //Obtain qdot
        for(i = 1; i < fluxcount ; i++ )
        {

        if( i == 1)
        {
                v_1Dgetval(flux,i-1,fluxcount,fluxL);
                v_1Dgetval(flux,i,fluxcount,fluxR);
        }else
                v_1Dgetval(flux,i,fluxcount,fluxR);

        //Obtain the derivative vector at node i
        qdot->v[i-chck1] = (fluxL->v[0] - fluxR->v[0]) / dx ;
        qdot->v[i+len-chck1] = (fluxL->v[1] - fluxR->v[1]) / dx ;
        qdot->v[i+2*len-chck1] = (fluxL->v[2] - fluxR->v[2]) / dx ;

        tmp = fluxL;
        fluxL = fluxR;
        fluxR = tmp; tmp = NULL;
}

//Apply boundary conditions

/*
*
* Left boundary condition
*
*/

//Obtain q = [rho, rho*u, rho*E ] values
//for use in obtaining spacial derivatives

if( myid == 0 )
{
```

```
/*Obtain values to do the
finite difference derivative
approximation*/
q11 = q->v[0];
q21 = q->v[len];
q31 = q->v[2*len];

q12 = q->v[1];
q22 = q->v[len+1];
q32 = q->v[2*len+1];

q13 = q->v[2];
q23 = q->v[len+2];
q33 = q->v[2*len+2];

//Obtain spacial derivatives of
//q = [rho, rho*u, rho*E ]
dq1dx = (-1.5*q11+2*q12-.5*q13)/dx;
dq2dx = (-1.5*q21+2*q22-.5*q23)/dx;
dq3dx = (-1.5*q31+2*q32-.5*q33)/dx;

//Convert q = [rho, rho*u, rho*E ]
//to v = [rho, u, p] on the first node
rho = q11;
u = q21/q11;
p = (gamma-1)*(q31-.5*q21*u);

//Obtain speed of sound quantities
c2 = p*gamma/rho;
c = sqrt(c2);

//Define temporary variable values for less of a mess
tmp1 = q21/q11;
tmp2 = gamma-1;

//Find dqdt
dq1dt = -dq2dx;
dq2dt = -( -tmp1*tmp1*(3-gamma)*dq1dx/2.0+
```

```
        (3-gamma)*dq2dx +
        (gamma-1)*dq3dx);
dq3dt = -( (tmp2*tmp1*tmp1*tmp1-gamma*q31*tmp1/q11)*dq1dx +
                (gamma*q31/q11 - 3*tmp2*tmp1*tmp1/2.0)*dq2dx
                + gamma*tmp1*dq3dx);

//Transform dqdt to dvdt
drhodt = dq1dt;
dudt = (-tmp1*dq1dt+dq2dt)/q11;
dpdt = (gamma-1)*(tmp1*tmp1*dq1dt/2.0 - tmp1*dq2dt + dq3dt);

//Transform dvdt to dwdt and apply boundary condition
dw1dt = drhodt - dpdt/c2;
dw2dt = (dpdt - rho*c*dudt)/2.0;
dw3dt = dw2dt;

//Rotate back from dwdt to dvdt
drhodt = dw1dt + (dw2dt + dw3dt)/c2;
dudt = (dw3dt - dw2dt)/(rho*c);
dpdt = dw2dt+dw3dt;

//Rotate back from dvdt to dqdt
dq1dt = drhodt;
dq2dt = u*drhodt + rho*dudt;
dq3dt = u*u*drhodt/2.0 +
        rho*u*dudt +
        dpdt/(gamma-1);

//Put resulting dqdt values into the solution vector
qdot->v[0] = dq1dt;
qdot->v[len] = dq2dt;
qdot->v[2*len] = dq3dt;
}

/*
 *
 * Right boundary condition
 *
```

```
 */

if( myid == numprocs-1 )
{
//Grab values for q for spacial derivative purposes
q11 = q->v[len-1];
q21 = q->v[2*len-1];
q31 = q->v[3*len-1];

q12 = q->v[len-2];
q22 = q->v[2*len-2];
q32 = q->v[3*len-2];

q13 = q->v[len-3];
q23 = q->v[2*len-3];
q33 = q->v[3*len-3];

//Obtain spacial derivatives for each q component
dq1dx = (1.5*q11-2*q12+.5*q13)/dx;
dq2dx = (1.5*q21-2*q22+.5*q23)/dx;
dq3dx = (1.5*q31-2*q32+.5*q33)/dx;

//obtain v = [rho, u, p] values
rho = q11;
u = q21/q11;
p = (gamma-1)*(q31-.5*q21*u);

//Obtain the speed of sound quantities
c2 = p*gamma/rho;
c = sqrt(c2);

//Define temporary variable values for less of a mess
tmp1 = q21/q11;
tmp2 = gamma-1;

//Obtain dqdt
dq1dt = -dq2dx;
dq2dt = -( -tmp1*tmp1*(3-gamma)*dq1dx/2.0+
```

```
        (3-gamma)*dq2dx +
        (gamma-1)*dq3dx);
dq3dt = -( (tmp2*tmp1*tmp1*tmp1-gamma*q31*tmp1/q11)*dq1dx +
                (gamma*q31/q11 - 3*tmp2*tmp1*tmp1/2.0)*dq2dx
                + gamma*tmp1*dq3dx);

//Obtain dvdt by transforming dqdt
drhodt = dq1dt;
dudt = (-tmp1*dq1dt+dq2dt)/q11;
dpdt = tmp2*(tmp1*tmp1*dq1dt/2.0 - tmp1*dq2dt + dq3dt);

//Obtain dqdt by transforming dvdt
//and apply boundary condition
dw1dt = drhodt - dpdt/c2;
dw3dt = (dpdt + rho*c*dudt)/2.0;
dw2dt = dw3dt;

//Rotate dwdt back to dvdt
drhodt = dw1dt + (dw2dt + dw3dt)/c2;
dudt = (dw3dt - dw2dt)/(rho*c);
dpdt = dw2dt+dw3dt;

//Rotate dvdt back to dqdt
dq1dt = drhodt;
dq2dt = u*drhodt + rho*dudt;
dq3dt = u*u*drhodt/2.0 + rho*u*dudt + dpdt/tmp2;

//Put resulting dqdt values into solution vector
qdot->v[len-1] = dq1dt;
qdot->v[2*len-1] = dq2dt;
qdot->v[3*len-1] = dq3dt;
}


/*
 *
 * Free memory
 *
```

```
        */
        v_free(flux);
        v_free(fluxL);
        v_free(fluxR);
}
```

References

[1] Bodony, D.J.,*AE 410 Spring 2012 Roe Scheme Notes*,AE 410 Notes, 2012.

[2] Gropp, W. and Lusk,E.L. and Skjellum, A., *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface*,The MIT Press, 1999.