# A Lot of Hot Air:
# Modeling Compressible Fluid Dynamics

New Mexico

Supercomputing Challenge

Final Report

April 4, 2007

Team Number 51

Los Alamos High School


Team Members

Jonathan Robey
Dov Shlachter

Teacher

Diane Medford

Project Mentor

Bob Robey

# Table of Contents:

**Executive Summary 3**

**1 Introduction 4**

*1.1 Problem Statement: 4*

*1.2 Objective 5*

*1.3 Background 6*

*1.4 Research 8*

*1.5 Innovation 9*

**2 Description 10**

*2.1 Methods 10*

*2.2 Mathematical Model 11*

*2.3 Computational Model: 14*

*2.4 Code Development: 16*

**3 Results 17**

*3.1 Test Problems 17*

3.1.1 The Sod Problem 18

3.1.2 Interacting Blast Waves 19

*3.2 Performance/Scaling 23*

3.2.1 Dual Core Laptop 23

3.2.2 LANL Supercomputer 25

**4 Conclusions 26**

*4.1 Teamwork 26*

*4.2 Model 26*

**5 Recommendations 26**

**References 28**

**Appendices 29**

*Appendix A-Code 29*

Main model 29
Display functions 46

**Executive Summary**:

The purpose of this project was to create a two dimensional model that can accurately predict the behavior of an ideal compressible fluid that follows the three conservation laws, that of energy, mass, and momentum. We achieved this by using a second order two-part method, known as the Lax-Wendroff method. We have overcome the boundary of the Gibbs phenomenon by incorporating TVD. Having tested the simulation against several benchmark problems, we can conclude that our model functions with a high degree of accuracy when the initial conditions do not legitimately create results similar to those created by the Gibbs phenomenon. However, as always, there are certain areas of the model that can be improved, including the possible addition of immovable reflective objects or "islands", taking friction of the interface into consideration, increasing the number of colors that the display function is capable of using, the writing of an input function to set up the initial conditions (for at the moment we must hard code in any initial changes we wish to make), and the breathtaking possibility of making the model fully three-dimensional.

The work was divided equally among us, with focus as to areas of expertise. What with this division of labor, the main programmer was able to create a working state-of-the-art model. The parallel processing was also brilliantly constructed, with a nearly perfect scaling factor. The division of work was really a great opportunity for one of us to read the scientific papers and truly get a grasp on the concept while the other one was getting some work done.

- 

# 1 Introduction:

## *1.1 Problem Statement:*

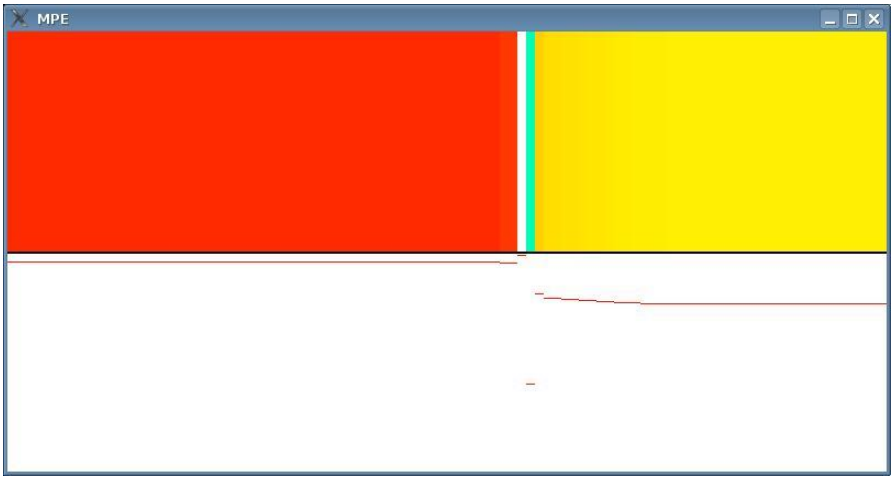
The modeling of compressible fluid dynamics has been a major field for many years, with very extensive work done by some very eminent people. One of the major problems with running fluid dynamics experiments is that the equipment involved, such as wind tunnels and shock tubes, are often very large, very expensive, or both. This has been a driving force in the movement to create accurate, versatile computational models for the compressible system of equations. Many different methods have been derived, some of which fit certain situations better than others. If accurate computer simulations can be made of these very complex systems, then those models can be applied to many different areas in science and engineering, such as meteorology, aerodynamics, design and construction of certain buildings, sound propagation, and Reimonn problems.

### *1.2 Objective*

We propose to create an accurate, second order, fully two dimensional program, capable of being tested against benchmark problems, and modeling the behavior of a compressible ideal gas. The model will be based on the three conservation laws, the conservation of mass, energy, and momentum, also known as the Euler equations. We will write this program to run on multiple processors, in order to run the program much more quickly while maintaining resolution, and we propose to create a display function to show the data in the form of colors.

### *1.3 Background*:

One of the most conventional tests of the nature of gasses is the shock tube. In a shock tube, there is a high-pressure area and a low-pressure area, separated by a thin membrane. The membrane is punctured, and the pressures are allowed to equalize. At the head of the shock, there is a sudden drop off point where the pressure goes very low very quickly, and first order equations are incapable of accurately depicting this, requiring six cells to resolve the shock. Second order equations, such as the Lax-Wendroff method, can deal with this drop more accurately, using only three cells to resolve the shock, but they have their own problems. Without TVD to stabilize it, this equation will develop the "Lax-Wendroff Wiggle," which is the predictor corrector dealing with radically different values for the half step and the whole step, causing the values of the variables to oscillate, sometimes leading to negative values. The computation does not deal well with negative numbers. Before enacting TVD, our program would always degenerate into this:




The dark region symbolizes areas where the computer is dealing with mathematical impossibilities, such as dividing by zero or taking the square root of

a negative number. These are commonly called not-a-numbers, or nans. TVD makes it so that the values do not oscillate, allowing the computer to display far more accurate results. However, even with TVD, the program is both memory and processor intensive and requires a lot of processing power. Simulations of gas behavior have always had to compromise between what type of supercomputer is available and how much accuracy is needed.

### *1.4 Research*

Many of the research papers that we perused to find methods and pointers were difficult to understand and difficult to implement. We did combine what we believe the best ideas to write an accurate two dimensional model. An upwind scheme was decided against because of the difficulty of implementing it in more than one dimension, and because it changes algorithm in the middle of the program. We did not use a Riemann solver because it would be difficult to apply, and the method does work without it in multiple dimensions. Also, in one dimension, having the computer do the Riemann solver increases the runtime of the model significantly, more so than is compensated by the gain of accuracy.

### *1.5 Innovation*

Our brilliance truly shines through when the production completion of a theoretical model is looked at. The program is very efficient, completing 5000 iterations for 1000 cells in about 44 seconds in one dimension, and terminating the calculations for a square 10000 cell 5000 iteration simulation in under a minute. The program is fully two dimensional, with no dimensional splitting, and displays the results in real time. Not including the display function, the entire program is less than one thousand lines long, allowing the processor to function smoothly and leading to a more efficient model.

# 2 Description

## *2.1 Methods*

Before we begin, it is necessary to define the difference between a mathematical model and a computational model. A mathematical model is a numerical representation of the real world, symbolized by the values of the different variables within each cell. A computational model is the set of equations or instructions that the computer uses to interpolate values of the variables when moving forward in time.

### *2.2 Mathematical Model*

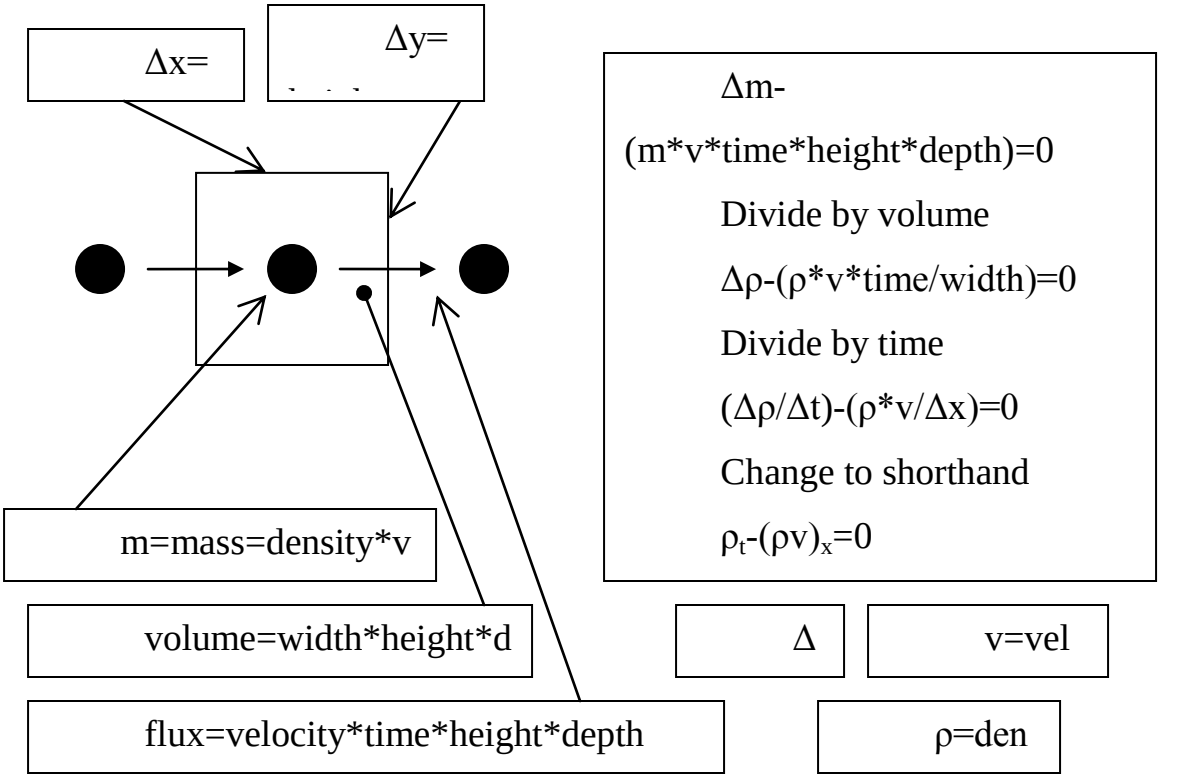
For the Euler equations, the mathematical model uses three [Note to Dov -3 with one more for each additional dimension] variable and the conservations thereof, represented by three equations. In each equation, the term with sub-'t' is the state variable, and the term with sub-'x' is the flux term. The fourth equation is the equation of state in terms of the other variables. The first part of the equation shows that we are using a gamma law gas in our simulation. The 'gamma' of a specific gas simply states, for an amount of energy dumped into the system, how much will be converted into kinetic energy (i.e., energy of movement), and how much will be transferred into internal energy (i.e., heat). The first three equations are known as the Euler equations, and have become fairly common in the realm of fluid dynamics.

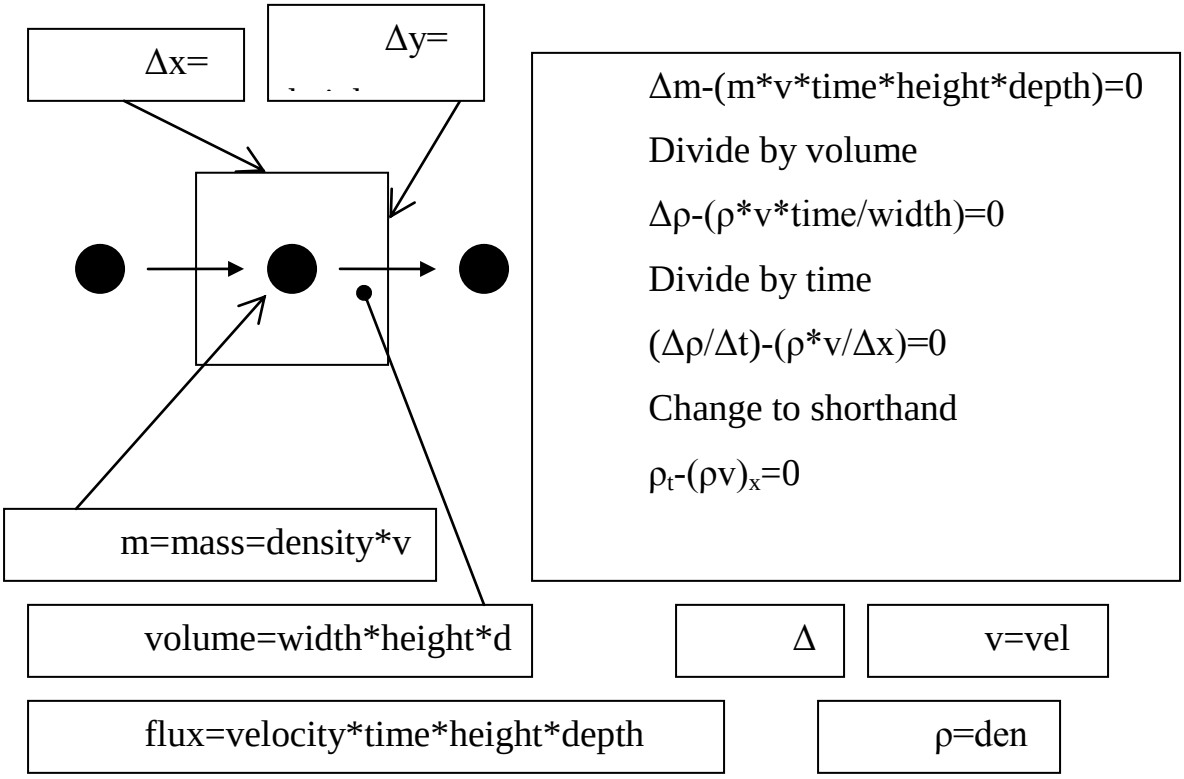
$$\rho_t + (\rho v)_x = 0$$ (Conservation of Mass)

$$(\rho v)_t + (\rho v^2 + p)_x = 0$$ (Conservation of Momentum)

$$E_t + [v(E + p)]_x = 0$$ (Conservation of Energy)

$$p = (\gamma - 1)[E - .5\rho v^2]$$ (Equation of State)


Δx=
Δy=
Δm-
(m*v*time*height*depth)=0
Divide by volume
Δρ-(ρ*v*time/width)=0
Divide by time
(Δρ/Δt)-(ρ*v/Δx)=0
Change to shorthand
ρt-(ρv)x=0
m=mass=density*v
volume=width*height*d
Δ
v=vel
flux=velocity*time*height*depth
ρ=den


Δx=
Δy=
Δm-(m*v*time*height*depth)=0
Divide by volume
Δρ-(ρ*v*time/width)=0
Divide by time
(Δρ/Δt)-(ρ*v/Δx)=0
Change to shorthand
ρt-(ρv)x=0
m=mass=density*v
volume=width*height*d
Δ
v=vel
flux=velocity*time*height*depth
ρ=den

### *2.3 Computational Model:*

We have decided to use an Eulerian model for the program, in which the modeled object moves throughout the mesh, instead of a Lagrangian method, in which the mesh moves with the object. Lagrangian is accurate in one dimension, but develops a problem known as mesh tangling in two dimensions, where the mesh becomes too twisted to be an accurate measurement. The method chosen was the Lax-Wendroff method, which is a second order predictor corrector systematic scheme. We chose it because second order methods have a high degree of accuracy. We did not choose an upwind scheme, because even though they are very accurate in one dimension[1], upwind schemes are much more difficult to use in multiple dimensions.

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2}}^{n+\frac{1}{2}} - F_{i-\frac{1}{2}}^{n+\frac{1}{2}}\right)$$

(First step)

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2}}^{n+\frac{1}{2}} - F_{i-\frac{1}{2}}^{n+\frac{1}{2}}\right)$$

(Second step)

TVD is a complex method of switching between first order and second order within the computational model. If TVD senses that a shock is imminent, TVD switches the computation to first order, which does not oscillate before and after

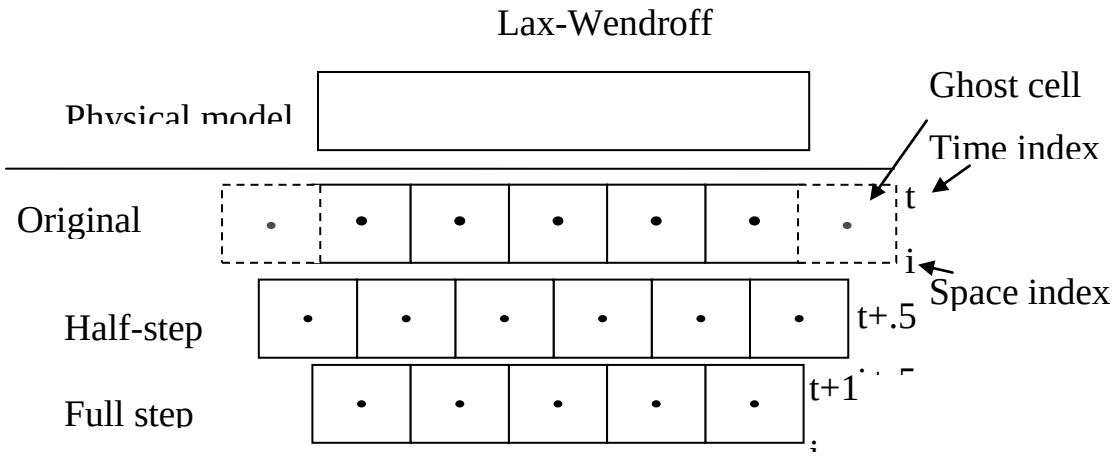
shocks, and then switching back again. The TVD stabilized method is as follows, adding this statement to the second part of the Lax-Wendroff method:

$$+\frac{1}{2}(\nu(1-\nu))[1-\phi(r^+,r^-)][U_{i+1}^n - U_i^n]$$

$$-\frac{1}{2}(\nu(1-\nu))[1-\phi(r^+,r^-)][U_i^n - U_{i-1}^n]$$

$$\nu = \frac{\lambda \Delta t}{\Delta x}$$

$$\phi(r^+,r^-) = \text{flux limiter}$$



### *2.4 Code Development:*

We have used our own code in the creation of this model, although we have observed how other programs implement certain details of the same basic procedures. The program is up and running on multiple processors using MPI, or Message Passing Interface, allowing larger meshes to be run more quickly and still enact the display function, as well as integrated parallel graphics.

For writing the code, we used the C language with the MPI and MPE libraries to add features we deemed to be necessary. We started by writing a one-dimensional version of the Lax-Wendroff method (around Thanksgiving). We found a large number of the flaws in the original program comparing our program's data for the Sod problem compared with a large quantity of published results. Over the winter holidays, we modified the program to allow two-dimensions to be implemented, which we debugged with the Shock-in-a-Box problem. We then incorporated the TVD scheme and fixed most of the major bugs in the program using both the Sod and Shock-in-a-Box problems. It was fairly easy to tell whether or not TVD had been incorporated correctly by the clarity of the running of the program. When it finally ran without bizarre colors and breaking points appearing in the display, we knew we had it down. In the last week of February, we fixed the few problems with the MPI scheme and had it run well, testing it with a simplified version of the Shock-in-a-Box problem. We did all the coding ourselves, however we did receive help with choosing the methods and debugging the program from our mentor.

- 11/26/06: Basic model completed with display functions.
- 12/23/06: TVD enabled
- 12/29/06, the working two dimensional model
- 2/27/07 MPI working

# 3 Results

## *3.1 Test Problems*

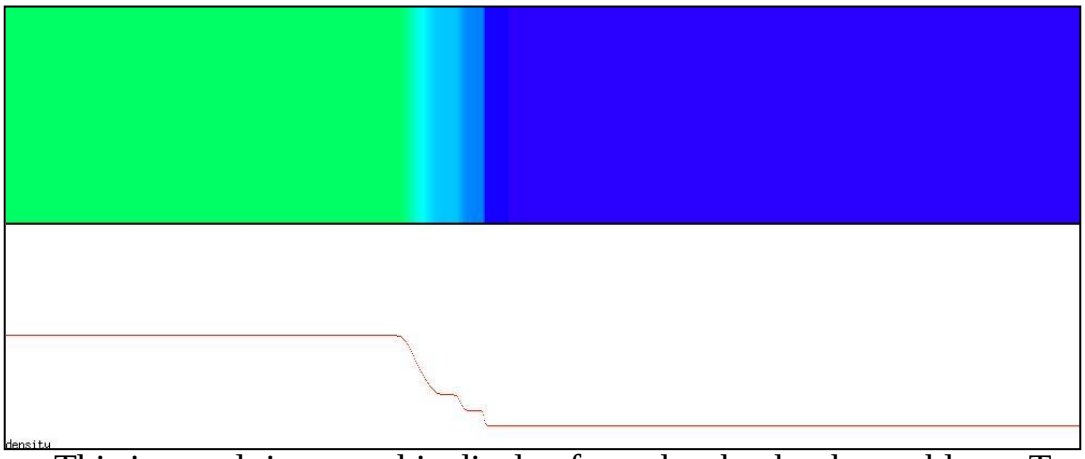
We have run extensive tests on our program using three test problems: the Sod problem, interacting blast waves, and shock-in-a-box. Each problem tests a different aspect of the program, and each presents its own difficulties, merits, and interesting challenges. Because of their one-dimensionality, we have also implemented a line graph that shows the magnitude of the value being displayed in y-coordinate for the shock tube and interacting blast waves. This function allows us to compare our results with published results when taken in the same scale. Only with the Sod problem is the dynamic runtime graph used to measure the accuracy of the data. With the other two problems, the data is printed to a file and then compared to the accepted results.

### 3.1.1 The Sod Problem

The Sod problem, mentioned in section 1.3, has initial conditions of two different pressure regions separated by a membrane. When the membrane is punctured, the two are allowed to come into contact with each other, creating a shockwave. The shock tube was used to test TVD, as well as the wisdom in choosing a second order method. The steep slope at the head of the shock tells us that our implementation of the Lax-Wendroff method has worked fairly well, and the fact that the program runs without an advancing bar of nans shows us that the TVD method has paid off.

$0 < x < 6$ density = 1 pressure = 1 momentum = 0

$6 < x < 10$ density = .125 pressure = .1 momentum = 0



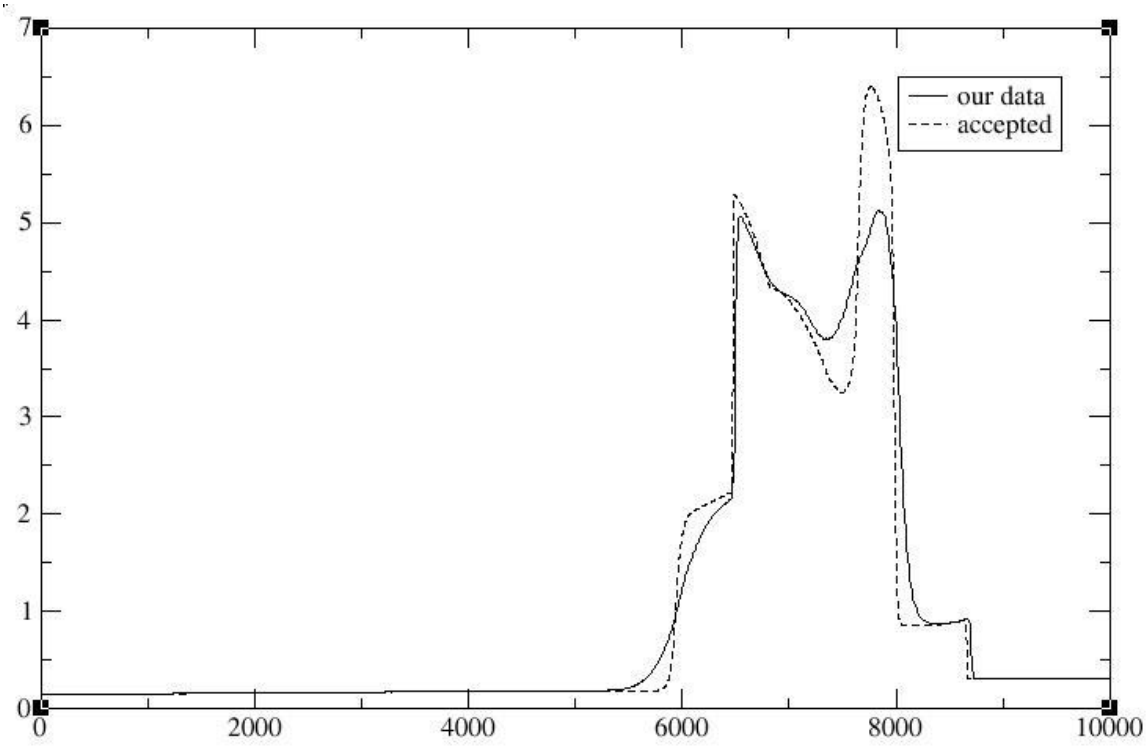

This is a real time graphic display from the shock tube problem. To derive the accepted results, one must go through the problem with an analytical solution, which we were not able to do. The results are published in numerous papers, including Gary Sod's paper. In those results, going from right to left, there is a flat line of low pressure, a vertical line leading to another flat line, and then a sloped line that levels out. As can be seen, we have almost perfectly vertical lines in our simulation as a result of using a second order method.

### 3.1.2 Interacting Blast Waves

The Interacting Blast Waves problem[2] is similar to the Sod problem, but with a slightly higher complexity. This problem starts with three regions, going from left to right: high pressure gas, low pressure gas, medium pressure gas.

| |
|---|
| $0.0<x<0.1$ rho=1.0 $m_x$=0.0 p=1000.0 |
| $0.1<x<0.9$ rho=1.0 $m_x$=0.0 p=0.01 |
| $0.9<x<1.0$ rho=1.0 $m_x$=0.0 p=100.0 |

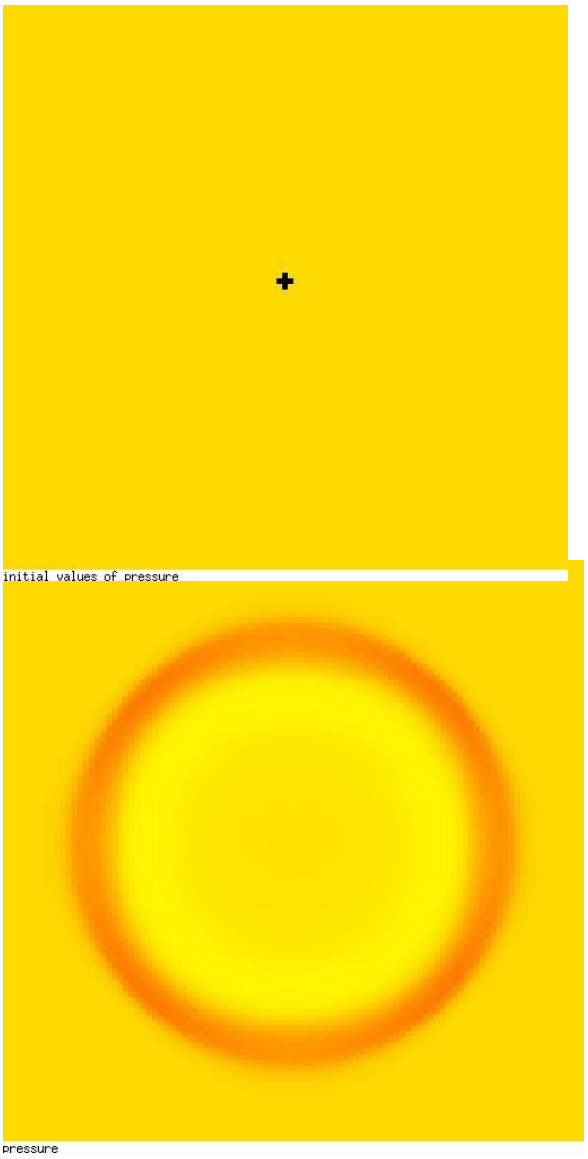
The blast waves converge at the right-hand side, because the left-hand region rushes to equalize with a force greater than that of the right-hand side gas. At the convergence of the two waves, there is a region in that, in the benchmark, has a very high resolution and very definite form. We measure the accuracy of our model by putting our model at the same scale and same precise initial conditions, and then comparing our results to the benchmark at the same point in time. Our results match well with the benchmark, although not perfectly, which shows us that our method returns accurate results. Due to the lack of an analytical soulution to this test problem, it is conventional to compare your model's results on a coarser mesh to the results on a finer mesh.

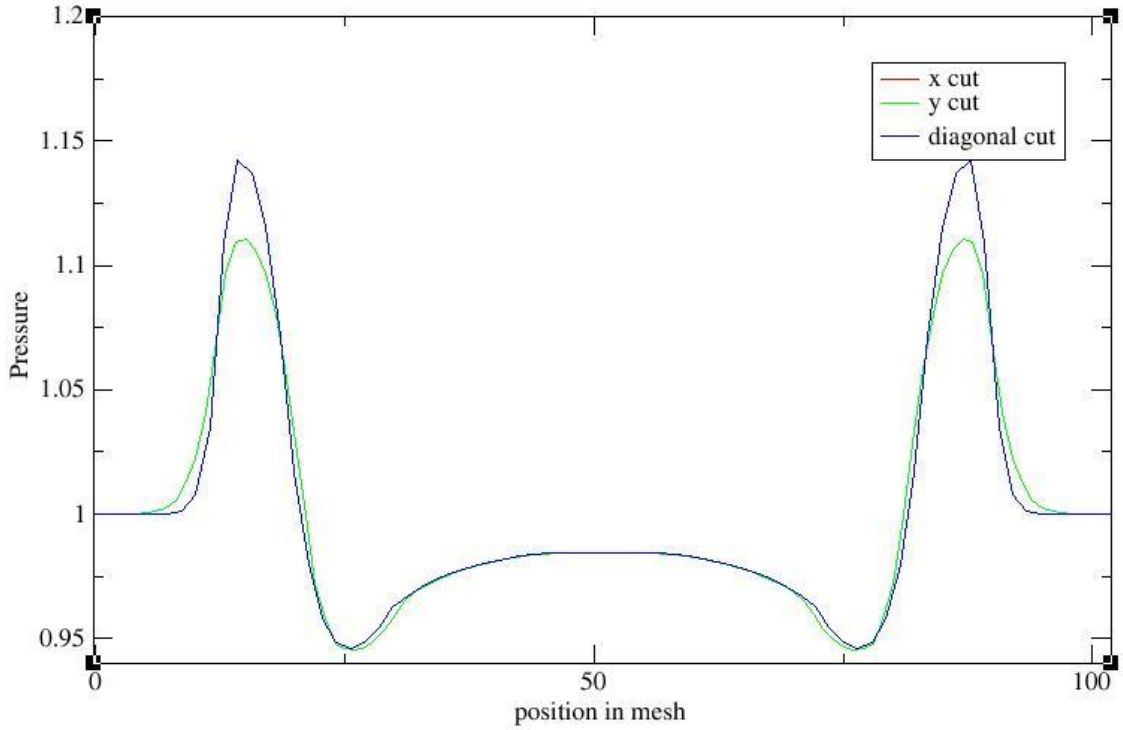



The above accepted results were generated using our model with a very fine mesh. We stretched the coarser results to be the same length as the fine results. Then, we superimposed the two data sets on top of each other to give us a clear representation of the accuracy of our coarse results.

### 3.1.3 Shock-in-a-Box

The shock-in-a-box problem is much more difficult to test against a benchmark because of its two dimensional nature, which renders displaying all of the data in a single line plot impossible, as well as providing the great challenge of making the program two dimensional. This problem starts with an equal pressure area surrounding a very dense and energetic core at time zero. At time > 0, the center expands in a (hopefully) spherical pattern. Our simulation has the shock spreading outwards in almost perfect symmetry, which shows that the model works well. However, due to the finite resolution that we are dealing with, the starting conditions form a plus sign instead of the preferable circle, and as a result of that there are areas of slightly higher pressure where the diagonal of the display intersect the sphere. This can be overcome with using a larger cell array, which is made practical with the application of multiple processor capabilities.



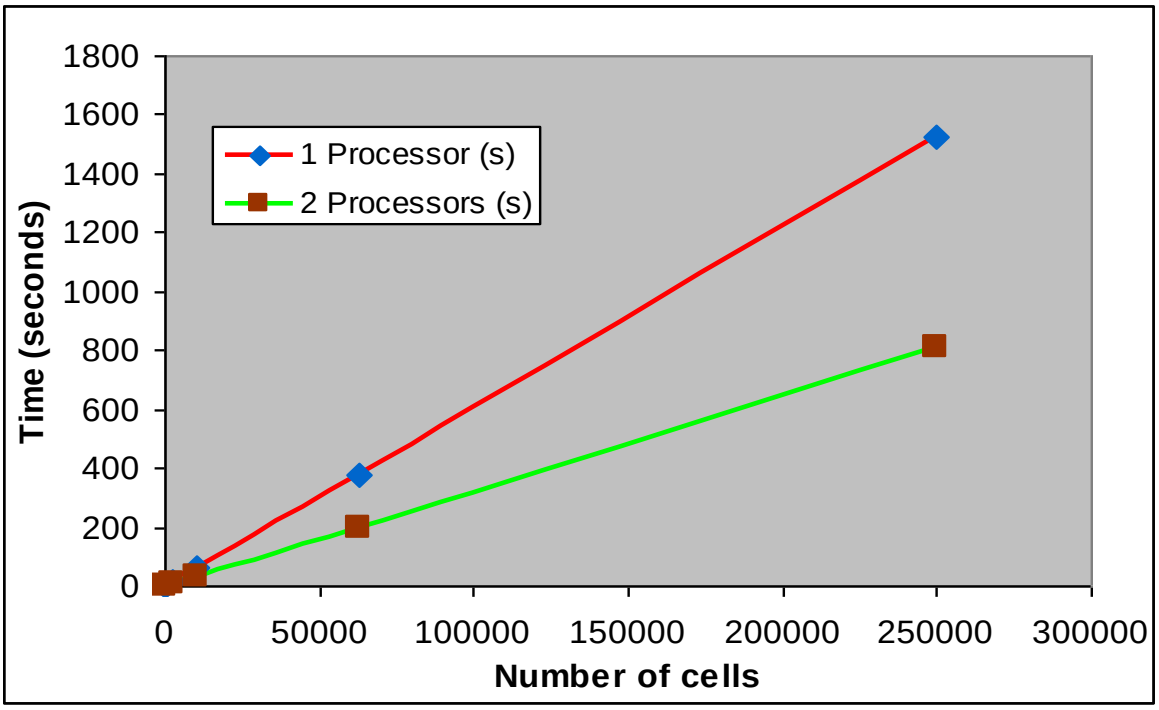



Above is a line graph of three specific rows of cells from the shock-in-a-box, the middle x row, the middle y row, and the straight diagonal row. As can be seen, the x and y rows are so identical that they can be superimposed, making them indistinguishable from one another. The diagonal is slightly imperfect due to the above mentioned higher pressure corners and finite resolution, but the effects are minimal. We also had to multiply the diagonal cut by the square root of two, because it its length was measured without taking into account the uses a straight line calculation of length, effectively giving the hypotenuse of a 45-45-90 right triangle.

## *3.2 Performance/Scaling*

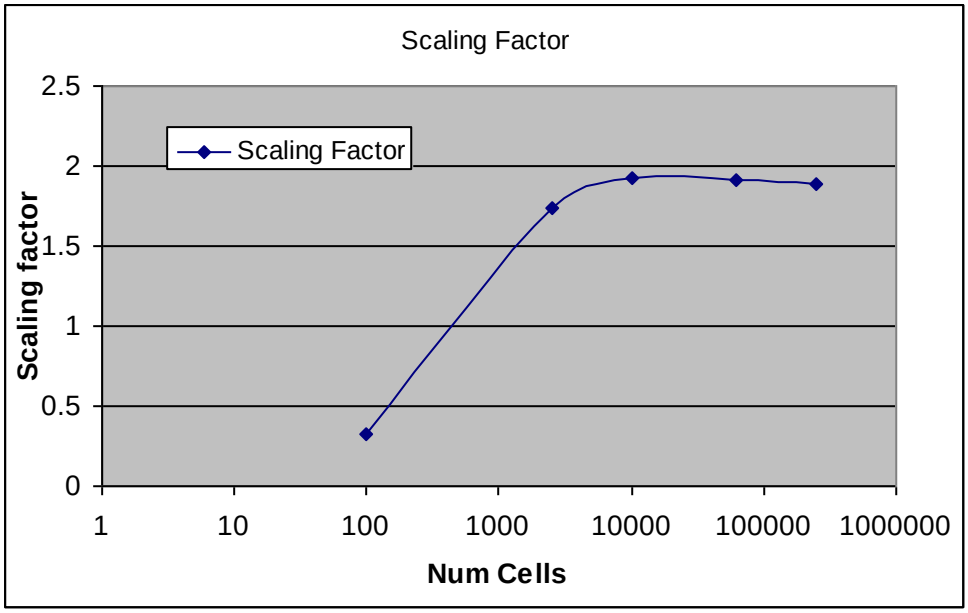
### 3.2.1 Dual Core Laptop

Due to questions of the efficiency of dual-core processors, we decided to test our model on a Hewlett-Packard dual core machine and an actual supercomputer. The two graphs below are, in order, the time it took for calculations to be done and the scaling factor, all on the Hewlett-Packard. When the number of cells was fairly small, the scaling factor was not much higher than one, and was even lower than one when tested at one hundred cells, presumably because the effort of transferring data and dividing the grid outweighed the effort of one processor simply doing the calculations.




The scaling factor approaches two when thousands or millions of cells are used in the array, but would presumably slowly decrease as the number of cells becomes incredibly large, possibly because of storing that much data in regular memory instead of cache. The cell array is divided using one dimensional static decomposition, in which the mesh is divided using parallel lines, and roughly equal

sections are given to each processor. So far, we have not written any code to weight the speed of a processor into the fraction of the mesh that it gets, which would be useful if vastly different processors were used within a supercomputer.




These tests were run with a 2 GHz Core 2 duo processor T7200 machine with 2.0 GB of RAM.

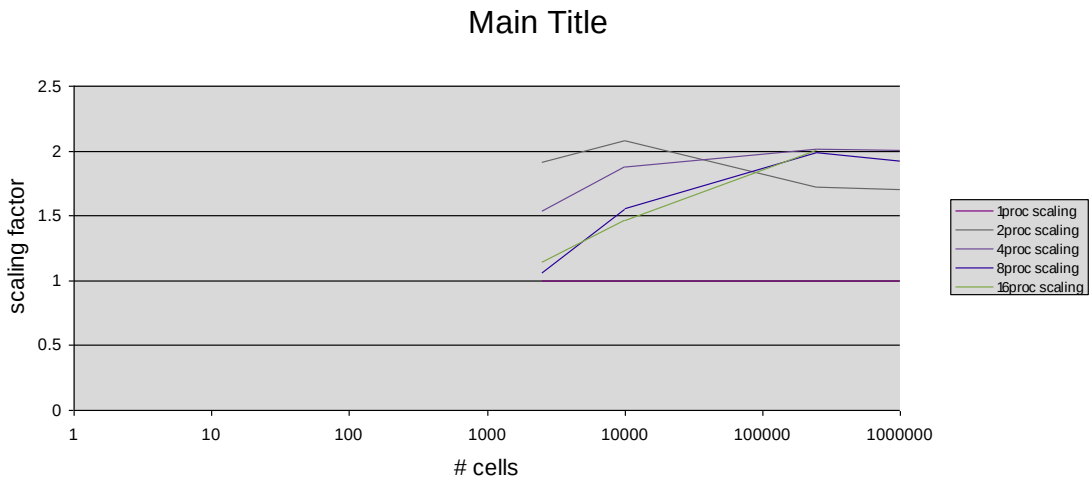
Very close to ideal scaling factor (2)

Communication may slow down as number of cells increases

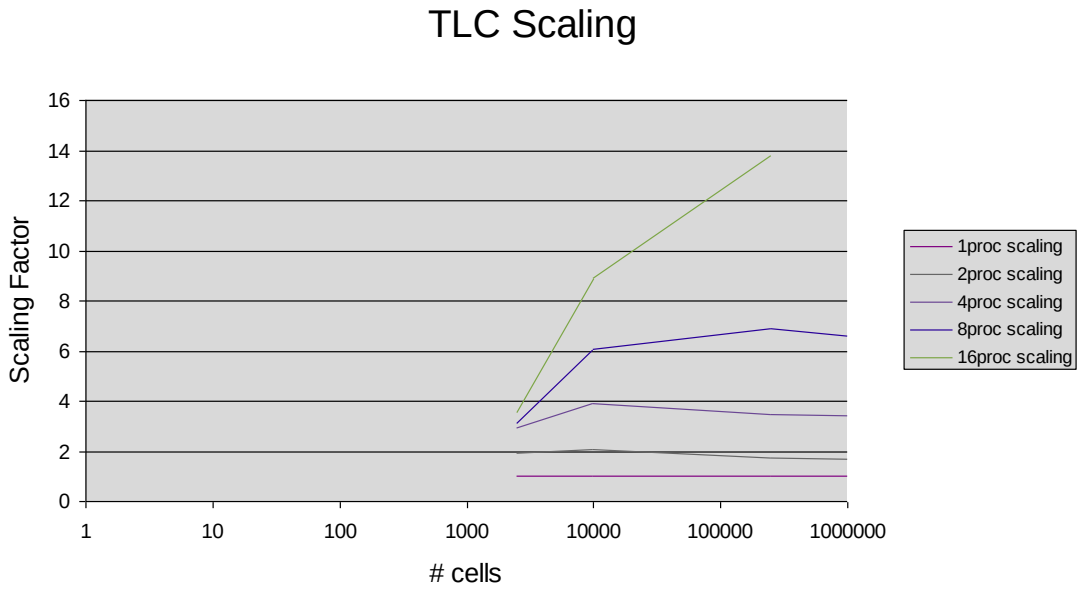
Need about 1000 cells per processor to offset communication

### 3.2.2 LANL Supercomputer

We also tested the program on the Turquoise Linux Cluster (TLC). The model performed very well in the tests. The model ran about twice as fast with twice as many processors every time. TLC is made of 220 AMD Opteron 2GHz processors each with a 1MB cache and 4GB of RAM per CPU.




Runtime as compared to runtime with half as many processors




Runtime as compared to runtime with 1 processor

# 4 Conclusions

## *4.1 Teamwork*

We developed our team with the knowledge of the work that needed to be accomplished and the knowledge of our particular skills. Our more technical person is a brilliant programmer and can quickly implement numerical methods into code, but can occasionally become obsessed with tiny details. The technical writer is an insightful mathematician and technical writer, but has difficulty grasping some of the more abstract research papers. This project was started from nothing, with a new team and a new project, although we hope to continue.

## *4.2 Model*

The model performs very well as seen by the test problems. In the two-dimensional version there are slight inaccuracies due to resolving the supposedly circular starting conditions to a plus. Due to the finite memory of the computer, there are slight round-off errors that grow as the program runs. The simulation runs very smoothly, but the visual display can get choppy when there are too many cells, and must be slowed down in order to be seen when there are too few cells. The multiprocessor code works well since with 2 processors it takes roughly half the time.

# 5 Recommendations

We could improve our model in a few ways. We could make it into a fully 3-dimensional model or we could add capability to calculate around immovable objects or “islands”. We could also try one of the better methods, add Multi-Material capability and apply our model to interesting problems

**Acknowledgements**

Robert W. Robey

For explaining theory and mentoring

Jack Shlachter

For help with equations and theory

Craig Rasmussen

For a recent version of Eclipse

Diane Medford

For sponsoring the Supercomputer Challenge

Supercomputing Consulter

For help in general

Thomas Laub

For critiquing the interim report

## References

[1]Rider, William J. "A Comparison of TVD Lax-Wendroff Methods." 1993.

[2]Paul Woodward and Philip Colella, "The Numerical Simulation of Two-Dimensional Fluid Flow with Strong Shocks", Journal of Computational Physics, 1984

S. F. Davis, "A Simplified TVD Finite Difference Scheme via Artificial Viscosity", SIAM J. Sci. Stat. Computing, Vol. 8, No. 1, January 1987, pp. 1-18

H. C. Yee, "Construction of Explicit and Implicit Symmetric TVD Schemes and Their Applications", Journal of Computational Physics, Vol. 68, 1987, pp 151-179.

Gary A. Sod, "A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws", Journal of Computational Physics, Vol. 27, 1978, pp 1-30.

Evan Scannapieco and Francis H. Harlow, Introduction to Finite-Difference Methods for Numerical Fluid Dynamics, LANL, 1995

Randall J. LeVeque, Numerical Methods for Conservation Laws, Birkhauser, 1992

Robert W. Robey, "Effects of Subscale Flow Structure On Numerical Simulation of Compressible Flows", Proceedings of the Thirteenth International Symposium on Military Applications of Blast Simulation (MABS-13)

# Appendices

## *Appendix A-Code*

### Main model

```
#include <mpi.h>
#define MPE_GRAPHICS
#include "mpe.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hydro.h"

/*************************************************************************
 *                                  SAPIENT
 *                Copyright 2007 Jonathan Robey, Dov Shlachter
 *                 Written for Supercomputing Challenge 2006-2007
 *                              ALL RIGHTS RESERVED
 *          SAPIENT is a proprietary program and is protected by copyright.
 *    Reproduction and use without permission is expressly prohibited.
 *
 *
*************************************************************************/

//define macro to find maximum of 3 values
#define MAX3(a,b,c) (max(max((a),(b)),(c)))
//define macro to find maximum of 2 values
#define max(f,g) ((f)>(g))?(f):(g)
//define macro to find minimum of 3 values
#define MIN3(a,b,c) (min(min((a),(b)),(c)))
//define macro to find minimum of 2 values
#define min(f,g) ((f)<(g))?(f):(g)
//define macro for squaring a number
#define SQ(x) ((x)*(x))

void display(int matrix_size_y, int matrix_size_x, double **temp, int
my_offset,
  int mysize, double maxscale);
void set_label(char *text);
void display_cond(int matrix_size_x, int matrix_size_y, double **temp,
   int my_offset, int mysize, double maxscale, char *text);
void get_cor(int *i, int *j, int my_offset, int NX, int NY);

int *ivector(int n);
int **imatrix(int m, int n);
double *dvector(int n);
double **dmatrix(int m, int n);
double ***dtrimatrix(int k, int m, int n);
double qxcalc(int i, int j, int ISHOCK);
double qycalc(int i, int j, int JMAX);
double (*qlimit)(double rminus, double rplus);
double qa(double rminus, double rplus);
double qb(double rminus, double rplus);
double qc(double rminus, double rplus);
```

```
double qd(double rminus, double rplus);
double qe(double rminus, double rplus);

  double   **rho, **mx, **my, **E, **p;        //state variables
  double   **rhoplusx, **rhoplusy, **mxplusx, **mxplusy, **myplusx,
**myplusy, **Eplusx, **Eplusy, **pplusx, **pplusy;//half-step arrays

double enginet(int matrix_size_x, int matrix_size_y, int ntimes, MPI_Comm
comm)
{
  FILE     *fp=NULL;
  char     filename[20];
  int      rank, size ;
  int      next, prev ;
  int      i, j, k;
  int      n;
  int      mysize;
  int      my_offset;
  int      maxind, ierr;                          //holder variable for
index
  int      *counts, *displs;
  double   *deltaX, *deltaY, **volume;         //size of cell
  double   nux, nuy, csqr, csx, csy, vx, q, cv, w;                      //
nu holders for second pass(TVD) and holder vars for calculations
  double   **temp;                             //holder array for display
  double   hold;                               //holder variables for finding
deltaT
  double   *sendbuf, *recvbuf;                 //communication buffers for
  double   deltaT, sigma, gamma, localmax, globalmax, maxDX;
//timestep, sigma, gamma, max for current processor, max for all
  double   maxScale;                           //
  double   time=0;                             //computer simulation time
  double   slavetime, totaltime, starttime ;   //variables to calculate time
taken for the program to run
//  double   eps = 1.0E-14;//variable for comparing/testing calculations that
will be affected by roundoff errors
//  double   rho_expected, mx_expected, E_expected, p_expected,
deltaT_expected;//variables for testing the program
  double   myTM, myTE, TotalMass, TotalEnergy, oldTM, oldTE;    //variables
for checking conservation of mass/energy
  double   drho, dmx, dmy, dE, **deltarho, **deltamx, **deltamy,
**deltaE;//storage variable for the changes in state varables in the second
pass
  char     *desc;           //variable for labels
  if(rank==0){printf("Copyright 2007\n");}

  sigma=0.5;
  qlimit=qb;//qa bad qd bad qe bad
  gamma=1.4;

  /* Determine size and my rank in MPI_COMM_WORLD communicator */
  MPI_Comm_size(MPI_COMM_WORLD, &size) ;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank) ;

  /* Set neighbors */
  if (rank == 0)
```

```
  prev = MPI_PROC_NULL;
else
  prev = rank-1;
if (rank == size - 1)
  next = MPI_PROC_NULL;
else
  next = rank+1;

/* Determine my part of the matrix */
mysize = matrix_size_y/size + ((rank < (matrix_size_y % size)) ? 1 : 0 ) ;
my_offset = rank * (matrix_size_y/size);
if (rank > (matrix_size_y % size)) my_offset += (matrix_size_y % size);
else                               my_offset += rank;

if(debug==1){printf("my rank is %d and mysize is %d\n", rank,mysize);}
/* allocate the memory dynamically for the matrix */
rho = dmatrix(mysize+4, matrix_size_x+2);
mx = dmatrix(mysize+4, matrix_size_x+2);
my = dmatrix(mysize+4, matrix_size_x+2);
E = dmatrix(mysize+4, matrix_size_x+2);
deltarho = dmatrix(mysize+2, matrix_size_x+2);
deltamx = dmatrix(mysize+2, matrix_size_x+2);
deltamy = dmatrix(mysize+2, matrix_size_x+2);
deltaE = dmatrix(mysize+2, matrix_size_x+2);
p = dmatrix(mysize+4, matrix_size_x+2);

rhoplusx = dmatrix(mysize, matrix_size_x+1);
mxplusx = dmatrix(mysize, matrix_size_x+1);
myplusx = dmatrix(mysize, matrix_size_x+1);
Eplusx = dmatrix(mysize, matrix_size_x+1);
pplusx = dmatrix(mysize, matrix_size_x+1);

rhoplusy = dmatrix(mysize+1, matrix_size_x);
mxplusy = dmatrix(mysize+1, matrix_size_x);
myplusy = dmatrix(mysize+1, matrix_size_x);
Eplusy = dmatrix(mysize+1, matrix_size_x);
pplusy = dmatrix(mysize+1, matrix_size_x);

temp = dmatrix(mysize+2, matrix_size_x+2);
volume = dmatrix(mysize+4, matrix_size_x+2);
deltaX = dvector(matrix_size_x+2) ;
deltaY = dvector(mysize+4) ;

counts = ivector(size);
displs = ivector(size);
MPI_Gather(&mysize,1,MPI_INT,counts,size,MPI_INT,0,MPI_COMM_WORLD);

displs[0]=0;
for(i=1;i<=rank;i++){
  displs[i]=displs[i-1]+counts[i-1];
}
sendbuf=dvector(matrix_size_x);
if(rank==0&&debug==1){printf("Memory allocated\n");}
/*initialize matrix*/
if(SOD_PROBLEM!=0){
for(j=0;j<=mysize+3;j++){
  for(i=0;i<=matrix_size_x+1;i++){
```

```
    rho[j][i]=0.1;
    mx[j][i]=0.0;
    my[j][i]=0.0;
    E[j][i]=0.1;
  }
}//initialization used to prevent nans
/*//Sod Problem comparisonTime=2.5
for(j=2; j<mysize+2; j++){
    for(i=1; i<matrix_size_x+2; i++){//mx set, first section
      mx[j][i]=0.0;
      my[j][i]=0.0;
      }
  for(i=1; i<(.6*matrix_size_x+2); i++){//E set, first section
    rho[j][i]=1.0;
      E[j][i]=2.5;
    }
  for(i=(.6*matrix_size_x+2);i<(matrix_size_x+2);i++){
    rho[j][i]=.125;
      E[j][i]=.25;
  }
}//*/
/*//Interacting Blast Waves comparisonTime=0.038
for(j=2; j<mysize+2; j++){
    for(i=1; i<matrix_size_x+2; i++){//mx set, first section
      rho[j][i]=1.0;
      mx[j][i]=0.0;
      my[j][i]=0.0;
      }
  for(i=1; i<(.1*matrix_size_x+2); i++){//E set, first section
      E[j][i]=2500.0;
    }
    for(i=(.1*matrix_size_x+2);i<(.9*matrix_size_x+2);i++){
      E[j][i]=0.025;
    }
  for(i=(.9*matrix_size_x+2);i<(matrix_size_x+2);i++){
    E[j][i]=250.0;
  }
}//*/
//2D init
for(j=0;j<=mysize+3;j++){
  for(i=0;i<=matrix_size_x+1;i++){
    rho[j][i]=1.0;
    mx[j][i]=0.0;
    my[j][i]=0.0;
    E[j][i]=2.5;
  }
}//*/
/*//1 Processor Shock-In-A-Box init
rho[52][51]=5.0;
E[52][51]=50.0;
rho[52][52]=5.0;
E[52][52]=50.0;
rho[51][51]=5.0;
E[51][51]=50.0;
rho[53][51]=5.0;
E[53][51]=50.0;
rho[52][50]=5.0;
```

```
  E[52][50]=50.0;//*/
  //Multi-Processor test init
  if(my_offset==0){
      rho[2][1]=5.0;
      E[2][1]=100.0;
  }//*/
  for(i=0; i<=matrix_size_x+1; i++){//deltaX set, first section
    deltaX[i]=0.1;
  }
  for(j=0; j<=mysize+3; j++){//deltaY set, first section
    deltaY[j]=0.1;
  }
  }
  if(rank==0&&debug==1){printf("initial values for state variables set\n");}
  /*initialize volume*/
  for(j=0;j<=mysize+3;j++){
      for(i=0;i<=matrix_size_x;i++){
            volume[j][i]=deltaX[i]*deltaY[j];
      }
  }
  //initialize pressure
  for(j=0;j<=mysize+3;j++){
    for(i=0;i<=matrix_size_x+1;i++){
      p[j][i]=(gamma-1.0)*(E[j][i]-
0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]));
      //if(rank==0){printf("i %3.3d j %3.3d rhait_time 0o %lf mx %lf my %lf E
%lf p %lf\n", i, j, rho[j][i], mx[j][i], my[j][i], E[j][i], p[j][i]);}
    }
  }
  if(rank==0&&debug==1){printf("initial values set\n");}
  //display initial values

  for(j=1;j<=mysize+2;j++){
    for(i=0;i<=matrix_size_x+1;i++){
      //Choose what information to display
      temp[j-1][i]=p[j][i];maxScale=2.0;desc="initial values of
pressure";//pressure
      //temp[j-1][i]=rho[j][i];maxScale=6.0;desc="initial values of
density";//density
      //temp[j-1][i]=E[j][i];maxScale=5.0;desc="initial values of
energy";//energy
      //temp[j-
1][i]=0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]);maxScale=0.25;desc="initial
values of kinetic energy";//kinetic energy
      //temp[j-1][i]=(E[j][i]-
0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]))/rho[j][i];maxScale=5.0;desc="ini
tial values of internal energy";//internal energy
      }
  }
  if(display_on==1){display_cond(matrix_size_x, matrix_size_y, temp,
my_offset, mysize, maxScale, desc);}
  if(rank==0&&debug==1){printf("initial values displayed\n");}
  /* run the simulation for given number of iterations */
  starttime = MPI_Wtime() ;
  for (n = 0; n < ntimes; n++) {
    MPI_Request      req[32];
    MPI_Status       status[32];
```

```
    /* Send and receive boundary information */
    MPI_Isend(rho[2],matrix_size_x+2,MPI_DOUBLE,prev,1,MPI_COMM_WORLD,req);

MPI_Irecv(rho[mysize+2],matrix_size_x+2,MPI_DOUBLE,next,1,MPI_COMM_WORLD,req+
1);
    MPI_Isend(rho[3],matrix_size_x+2,MPI_DOUBLE,prev,2,MPI_COMM_WORLD,req+2);

MPI_Irecv(rho[mysize+3],matrix_size_x+2,MPI_DOUBLE,next,2,MPI_COMM_WORLD,req+
3);

MPI_Isend(rho[mysize],matrix_size_x+2,MPI_DOUBLE,next,3,MPI_COMM_WORLD,req+4)
;
    MPI_Irecv(rho[0],matrix_size_x+2,MPI_DOUBLE,prev,3,MPI_COMM_WORLD,req+5);

MPI_Isend(rho[mysize+1],matrix_size_x+2,MPI_DOUBLE,next,4,MPI_COMM_WORLD,req+
6);
    MPI_Irecv(rho[1],matrix_size_x+2,MPI_DOUBLE,prev,4,MPI_COMM_WORLD,req+7);
    if(rank==0&&debug==1){printf("values for rho communicated\n");}
    MPI_Isend(mx[2],matrix_size_x+2,MPI_DOUBLE,prev,5,MPI_COMM_WORLD,req+8);

MPI_Irecv(mx[mysize+2],matrix_size_x+2,MPI_DOUBLE,next,5,MPI_COMM_WORLD,req+9
);
    MPI_Isend(mx[3],matrix_size_x+2,MPI_DOUBLE,prev,6,MPI_COMM_WORLD,req+10);

MPI_Irecv(mx[mysize+3],matrix_size_x+2,MPI_DOUBLE,next,6,MPI_COMM_WORLD,req+1
1);

MPI_Isend(mx[mysize],matrix_size_x+2,MPI_DOUBLE,next,7,MPI_COMM_WORLD,req+12)
;
    MPI_Irecv(mx[0],matrix_size_x+2,MPI_DOUBLE,prev,7,MPI_COMM_WORLD,req+13);

MPI_Isend(mx[mysize+1],matrix_size_x+2,MPI_DOUBLE,next,8,MPI_COMM_WORLD,req+1
4);
    MPI_Irecv(mx[1],matrix_size_x+2,MPI_DOUBLE,prev,8,MPI_COMM_WORLD,req+15);
    if(rank==0&&debug==1){printf("values for mx communicated\n");}
    MPI_Isend(my[2],matrix_size_x+2,MPI_DOUBLE,prev,9,MPI_COMM_WORLD,req+16);

MPI_Irecv(my[mysize+2],matrix_size_x+2,MPI_DOUBLE,next,9,MPI_COMM_WORLD,req+1
7);

MPI_Isend(my[3],matrix_size_x+2,MPI_DOUBLE,prev,10,MPI_COMM_WORLD,req+18);

MPI_Irecv(my[mysize+3],matrix_size_x+2,MPI_DOUBLE,next,10,MPI_COMM_WORLD,req+
19);

MPI_Isend(my[mysize],matrix_size_x+2,MPI_DOUBLE,next,11,MPI_COMM_WORLD,req+20
);

MPI_Irecv(my[0],matrix_size_x+2,MPI_DOUBLE,prev,11,MPI_COMM_WORLD,req+21);

MPI_Isend(my[mysize+1],matrix_size_x+2,MPI_DOUBLE,next,12,MPI_COMM_WORLD,req+
22);

MPI_Irecv(my[1],matrix_size_x+2,MPI_DOUBLE,prev,12,MPI_COMM_WORLD,req+23);
    if(rank==0&&debug==1){printf("values for my communicated\n");}
    MPI_Isend(E[2],matrix_size_x+2,MPI_DOUBLE,prev,13,MPI_COMM_WORLD,req+24);
```

```
MPI_Irecv(E[mysize+2],matrix_size_x+2,MPI_DOUBLE,next,13,MPI_COMM_WORLD,req+2
5);
    MPI_Isend(E[3],matrix_size_x+2,MPI_DOUBLE,prev,14,MPI_COMM_WORLD,req+26);

MPI_Irecv(E[mysize+3],matrix_size_x+2,MPI_DOUBLE,next,14,MPI_COMM_WORLD,req+2
7);

MPI_Isend(E[mysize],matrix_size_x+2,MPI_DOUBLE,next,15,MPI_COMM_WORLD,req+28)
;
    MPI_Irecv(E[0],matrix_size_x+2,MPI_DOUBLE,prev,15,MPI_COMM_WORLD,req+29);

MPI_Isend(E[mysize+1],matrix_size_x+2,MPI_DOUBLE,next,16,MPI_COMM_WORLD,req+3
0);
    MPI_Irecv(E[1],matrix_size_x+2,MPI_DOUBLE,prev,16,MPI_COMM_WORLD,req+31);
    if(rank==0&&debug==1){printf("values for E communicated\n");}
    MPI_Waitall(32, req, status);
    if(rank==0&&debug==1)printf("Communication succesful\n");
    //set boundry conditons
    for(j=2;j<=mysize+1;j++){
      rho[j][0]=rho[j][1];
        mx[j][0]=-mx[j][1];
      my[j][0]=my[j][1];
      E[j][0]=E[j][1];
        rho[j][matrix_size_x+1]=rho[j][matrix_size_x];
        mx[j][matrix_size_x+1]=-mx[j][matrix_size_x];
      my[j][matrix_size_x+1]=my[j][matrix_size_x];
        E[j][matrix_size_x+1]=E[j][matrix_size_x];
      }
    for(i=0;i<=matrix_size_x+1;i++){
      if(my_offset==0){
      rho[1][i]=rho[2][i];
        mx[1][i]=mx[2][i];
        my[1][i]=-my[2][i];
          E[1][i]=E[2][i];
      }
      if(matrix_size_y==my_offset+mysize){
        rho[mysize+2][i]=rho[mysize+1][i];
        mx[mysize+2][i]=mx[mysize+1][i];
        my[mysize+2][i]=-my[mysize+1][i];
        E[mysize+2][i]=E[mysize+1][i];
      }
    }
    if(rank==0&&debug==1)printf("Boundry conditions set\n");
    /*set pressure*/
    for(j=0;j<=mysize+3;j++){
       for(i=0;i<=matrix_size_x+1;i++){
              //p[j][i]=(gamma-1.0)*(E[j][i]-
0.5*((mx[j][i]+my[j][i])/rho[j][i]));
              p[j][i]=(gamma-1.0)*(E[j][i]-
0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]));
              //if(rank==0){printf("i %3.3d j %3.3d rho %lf mx %lf my %ulf E
%lf p %lf\n", i, j, rho[j][i], mx[j][i], my[j][i], E[j][i], p[j][i]);}
              //printf("%d %d %lf\n",i,j,p[j][i]);
       }
    }
    //set timestep
```

```
    localmax=0;
    for(j=1;j<=mysize;j++){
      for(i=1;i<=matrix_size_x;i++){
        hold=(fabs(mx[j][i]/rho[j][i])+sqrt(gamma*(p[j][i]/rho[j][i])));
        if(hold>localmax){
          maxind=i;
          localmax=hold;
        }
      }
    }
    MPI_Allreduce(&localmax, &globalmax, size, MPI_DOUBLE, MPI_MAX,
MPI_COMM_WORLD);
    maxDX=deltaX[maxind];//temporary solution for constant size of cells, for
variable sizeed cells a line of code is needed here to find the deltaX for
the maximum
      deltaT=(sigma*maxDX)/globalmax;
      time+=deltaT;
    if(rank==0&&debug==1){printf("deltaT set to %20.17f\n",deltaT);}

      /*if(rank==0){
            printf("iteration %d:\n", n);
            for(j=3;j<mysize+3;j++){
              for(i=1;i<matrix_size_x+1;i++){
                  printf("ghost cell[%d][%d]: %f\n", j, i, p[j][i]);
              }
            }
      }

      if(rank==1){
            printf("iteration %d:\n", n);
            for(j=3;j<mysize+3;j++){
              for(i=1;i<matrix_size_x+1;i++){
                  printf("real cell[%d][%d]: %f\n", j, i, p[j][i]);
              }
            }
      }//*/

/*
    for (j = 0; j < mysize; j++) {
      for (i = 1; i <= matrix_size_x; i++) {
          //Pressure calculation
        //p[j+1][i]=(gamma-1.0)*(E[j+1][i]-
0.5*((mx[j+1][i]*mx[j+1][i])/rho[j+1][i]));
        p[j+1][i]=(gamma-1.0)*(E[j+1][i]-
0.5*(SQ(mx[j+1][i])+SQ(my[j+1][i])/rho[j+1][i]));
      } // i loop
    } // j loop
*/
    /* For each element of the matrix ... */
    if (rank == 0&&debug==1) {printf("Before 1st pass\n");}
    //first pass
    //x direction
    for (j = 0; j < mysize; j++) {
      for (i = 0; i<=matrix_size_x;i++) {
            //density calculation
            rhoplusx[j][i]=0.5*(rho[j+2][i+1]+rho[j+2][i])-
(deltaT/(2.0*deltaX[i]))*(mx[j+2][i+1]-mx[j+2][i]);
```

```
            //momentum x calculation
            mxplusx[j][i]=0.5*(mx[j+2][i+1]+mx[j+2][i  ])-
(deltaT/(2.0*deltaX[i]))*

(((mx[j+2][i+1]*mx[j+2][i+1]/rho[j+2][i+1])+p[j+2][i+1])
                        -((mx[j+2][i  ]*mx[j+2][i  ]/rho[j+2][i  ])+p[j+2][i
]));
        //momentum y calculation
        myplusx[j][i]=0.5*(my[j+2][i+1]+ my[j+2][i  ])-
(deltaT/(2.0*deltaX[i]))*
                         ((mx[j+2][i+1]*(my[j+2][i+1]/rho[j+2][i+1]))
                         -(mx[j+2][i  ]*(my[j+2][i  ]/rho[j+2][i  ])));
            //Energy calculation;
            Eplusx[j][i]=0.5*(E[j+2][i+1]+E[j+2][i])-(deltaT/(2.0*deltaX[i]))
                  *(((mx[j+2][i+1]/rho[j+2][i+1])*(E[j+2][i+1]+p[j+2][i+1]))
                   -((mx[j+2][i  ]/rho[j+2][i  ])*(E[j+2][i  ]+p[j+2][i
])));
        //pressure calculation
        pplusx[j][i]=(gamma-1.0)*(Eplusx[j][i]-
0.5*(SQ(mxplusx[j][i])+SQ(myplusx[j][i]))/rhoplusx[j][i]);
        //printf("1st pass x direction iter %d i %d j %d\n", n, i, j);
        //printf("i %d j %d rhoplusx %lf mxplusx %lf myplusx %lf Eplusx %lf
pplusx %lf\n", i, j,
        //   rhoplusx[j][i], mxplusx[j][i], myplusx[j][i], Eplusx[j][i],
pplusx[j][i]);
      }
    }
    if(rank==0&&debug==1){printf("First pass x direction complete\n");}
    //y direction
    for(j = 0; j<= mysize; j++){
      for(i = 0; i<matrix_size_x;i++){
      //density calculation
      rhoplusy[j][i]=0.5*(rho[j+2][i+1]+rho[j+1][i+1])-
(deltaT/(2.0*deltaY[j+1]))*(my[j+2][i+1]-my[j+1][i+1]);
            //momentum x calculation
            mxplusy[j][i]=0.5*(mx[j+2][i+1]+ mx[j+1][i+1])-
(deltaT/(2.0*deltaY[j+1]))*
                         ((mx[j+2][i+1]*(my[j+2][i+1]/rho[j+2][i+1]))
                         -(mx[j+1][i+1]*(my[j+1][i+1]/rho[j+1][i+1])));
        //momentum y calculation
        myplusy[j][i]=0.5*(my[j+2][i+1]+my[j+1][i+1])-
(deltaT/(2.0*deltaY[j+1]))*

(((my[j+2][i+1]*my[j+2][i+1]/rho[j+2][i+1])+p[j+2][i+1])
                        -
((my[j+1][i+1]*my[j+1][i+1]/rho[j+1][i+1])+p[j+1][i+1]));
            //Energy calculation;#define printIterInfo 0
            Eplusy[j][i]=0.5*(E[j+2][i+1]+E[j+1][i+1])-
(deltaT/(2.0*deltaY[j+1]))
                  *(((my[j+2][i+1]/rho[j+2][i+1])*(E[j+2][i+1]+p[j+2][i+1]))
                   -
((my[j+1][i+1]/rho[j+1][i+1])*(E[j+1][i+1]+p[j+1][i+1])));
        //pressure calculation
        pplusy[j][i]=(gamma-1.0)*(Eplusy[j][i]-
0.5*((SQ(mxplusy[j][i])+SQ(myplusy[j][i]))/rhoplusy[j][i]));
```

```
        //printf("i %d j %d rhoplusy %lf mxplusy %lf myplusy %lf Eplusy %lf
pplusy %lf\n", i, j, rhoplusy[j][i], mxplusy[j][i], myplusy[j][i],
Eplusy[j][i], pplusy[j][i]);
      //printf("1st pass y direction iter %d i %d j %d\n", n, i, j);
      }
    }
    if(rank==0&&debug==1){printf("First pass complete\n");}
    /*for(j=2;j<=mysize+1;j++){
      for(i=1;i<=matrix_size_x;i++){
            if (isnan(rho[j][i])) {
              printf("Error -- rho[%d][%d]=%f\n",i,j,rho[j][i]);
            }
        if (isnan(E[j][i])) {
              printf("Error -- E[%d][%d]=%f\n",i,j,E[j][i]);
            }
    }
  }//*/
    //second pass
    for (j = 0; j <= mysize+1; j++) {
      for (i = 0; i <= matrix_size_x+1; i++) {
            //zero out deltas
            deltarho[j][i]=0.0;
            deltamx[j][i]=0.0;
            deltamy[j][i]=0.0;
            deltaE[j][i]=0.0;
      }
    }
    if(rank==0&&debug==1){printf("Deltas zeroed\n");}
    //x direction
    for (j = 0; j < mysize; j++) {
      for (i = 0; i<matrix_size_x+1;i++) {
            //printf("mxplusx %f\n",mxplusx[j][i]);
            vx=mxplusx[j][i]/rhoplusx[j][i];
            csx=sqrt(gamma*pplusx[j][i]/rhoplusx[j][i]);
            nux=(fabs(vx)+csx)*deltaT/deltaX[i];
            q=qxcalc(i, j+2, matrix_size_x);
            cv=nux*(1.0-nux);
            w=0.5*cv*(1.0-q);
            /*if (isnan(w)||n==3){
          printf("x i %d, j % d, cv %f, q %f w %f\n", i, j, cv, q, w);
          printf(" cv %f mxplusx %f rhoplusx %f pplusx %f\n",cv,
myplusx[j][i], rhoplusx[j][i], pplusx[j][i]);
          printf("vx %f csx %f\n",vx, csx);
            }//*/
            //w=0.0;
            //printf("rho[%d][%d]=%lf deltaT/deltaX[i] %lf mxplus %lf %lf
delta %lf w%lf\n",i,j,rho[j+1][i],
            //  deltaT/deltaX[i], mxplusx[j][i-1],mxplusx[j][i],
(deltaT/deltaX[i])*(mxplusx[j][i-1]-mxplusx[j][i]), w);
            //density calculation
            drho = (deltaT/deltaX[i])*mxplusx[j][i]-w*(rho[j+2][i+1]-
rho[j+2][i]);
            deltarho[j+1][i]  -=drho;
            deltarho[j+1][i+1]+=drho;
            //momentum x calculation
```

```
            dmx =
(deltaT/deltaX[i])*mxplusx[j][i]*mxplusx[j][i]/rhoplusx[j][i]-
w*(mx[j+2][i+1]-mx[j+2][i]);
            //printf("X PASS DD i %d j %d dmx %f mxplusx %f rhoplusx %f w %f
mx %f %f\n", i, j, dmx,
            //   mxplusx[j][i], rhoplusx[j][i], w, mx[ j+1][i+1],
mx[j+1][i]);
            deltamx[j+1][i]  -=dmx;
            deltamx[j+1][i+1]+=dmx;
            //momentum y calculation
        dmy = (deltaT/deltaX[i])*mxplusx[j][i]*myplusx[j][i]/rhoplusx[j][i]-
w*(my[j+2][i+1]-my[j+2][i]);
        deltamy[j+1][i]  -=dmy;
        deltamy[j+1][i+1]+=dmy;
            //Energy calculation
            dE =
(deltaT/deltaX[i])*(mxplusx[j][i]/rhoplusx[j][i])*(Eplusx[j][i]+pplusx[j][i])
-w*(E[j+2][i+1]-E[j+2][i]);
            deltaE[j+1][i]  -=dE;
            deltaE[j+1][i+1]+=dE;
            //printf("2nd pass x direction iter %d - i %d j %d + i %d j
%d\n", n, i, j+1, i+1, j+1);
      }
    }
    if(rank==0&&debug==1){printf("Second pass x direction complete\n");}
    //y direction
    for (j = 0; j < mysize+1; j++) {
      for (i = 0; i< matrix_size_x;i++){
            vx=myplusy[j][i]/rhoplusy[j][i];
            csx=sqrt(gamma*pplusy[j][i]/rhoplusy[j][i]);
            nuy=(fabs(vx)+csx)*deltaT/deltaY[j];
            q=qycalc(i+1, j+1, mysize);
            cv=nuy*(1.0-nuy);
            w=0.5*cv*(1.0-q);
            /*if (isnan(w)||n==3){
         printf("y i %d, j% d, cv %f, q %f w %f\n", i, j, cv, q, w);
        printf(" cv %f myplusy %f rhoplusy %f pplusy %f\n",cv,
myplusy[j][i], rhoplusy[j][i], pplusy[j][i]);
          printf("vy %f csy %f\n",vx, csx);
            }//*/
            //w=0.0;
            //printf("rho[%d][%d]=%lf deltaT/deltaX[i] %lf mxplus %lf %lf
delta %lf w%lf\n",i,j,rho[j+1][i],
            //  deltaT/deltaX[i], mxplusx[j][i-1],mxplusx[j][i],
(deltaT/deltaX[i])*(mxplusx[j][i-1]-mxplusx[j][i]), w);
            //density calculation
            drho = (deltaT/deltaY[j])*myplusy[j][i]-w*(rho[j+2][i+1]-
rho[j+1][i+1]);
            deltarho[j][i+1]  -=drho;
            deltarho[j+1][i+1]+=drho;
            //momentum x calculation
            dmx =
(deltaT/deltaY[j])*myplusy[j][i]*mxplusy[j][i]/rhoplusy[j][i]-
w*(mx[j+2][i+1]-mx[j+1][i+1]);
            //printf("YPASS DD i %d j %d dmx %f myplusy %f mxplusy %f
rhoplusy %f w %f mx %f %f\n", i, j,
```

```
            //  /home/jon/workspacedmx, myplusy[j][i], mxplusy[j][i],
rhoplusy[j][i], w, mx[j+1][i+1], mx[j][i+1]);
            deltamx[j][i+1]  -=dmx;
            deltamx[j+1][i+1]+=dmx;
            //momentum y calculation
        dmy = (deltaT/deltaY[j])*myplusy[j][i]*myplusy[j][i]/rhoplusy[j][i]-
w*(my[j+2][i+1]-my[j+1][i+1]);
        deltamy[j][i+1]  -=dmy;
        deltamy[j+1][i+1]+=dmy;
            //Energy calculation
            dE =
(deltaT/deltaY[j])*myplusy[j][i]/rhoplusy[j][i]*(Eplusy[j][i]+pplusy[j][i])-
w*(E[j+2][i+1]-E[j+1][i+1]);
            deltaE[j][i+1]  -=dE;
            deltaE[j+1][i+1]+=dE;
            //printf("2nd pass y direction iter %d - i %d j %d + i %d j
%d\n", n, i+1, j, i+1, j+1);
      }
    }
    if (rank == 0&&debug==1) {printf("Second pass y direction complete\n");}
    for (j = 1; j <= mysize+2; j++) {
      for (i = 0; i <= matrix_size_x+1; i++) {
            //add flux term to state variables
            rho[j][i]+=deltarho[j-1][i];
            mx[j][i]+=deltamx[j-1][i];
            my[j][i]+=deltamy[j-1][i];
            E[j][i]+=deltaE[j-1][i];
            //printf("rank %d adding flux term from i %d j %d to state
variables at i %d j%d\n", rank, i, j-1, i, j);
      }
    }
    if(rank==0&&debug==1){printf("flux term added\n");}
    for (j = 2; j < mysize+2; j++) {
      for (i = 1; i <= matrix_size_x; i++) {
            //pressure term only
            mx[j][i]-=(deltaT/deltaX[i])*( pplusx[j-2][i]-pplusx[j-2][i-1] );
            //printf("DD i %d j %d delta %f pplusx %f %f\n", i, j,
            //  -(deltaT/deltaX[i])*( pplusx[j-2][i]-pplusx[j-2][i-1] ),
          //  pplusx[j-2][i], pplusx[j-2][i-1]);
            my[j][i]-=(deltaT/deltaY[j])*( pplusy[j-1][i-1]-pplusy[j-2][i-1]
);
            //printf("DD i %d j %d delta %f pplusy %f %f\n", i, j,
            //  -(deltaT/deltaY[j])*( pplusy[j-1][i-1]-pplusy[j-2][i-1] ),
            //  pplusy[j-1][i-1], pplusy[j-2][i-1]);

            //Pressure calculation
        p[j][i]=(gamma-1.0)*(E[j][i]-
0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]));
        //if(rank==0){printf("rank %d n %3.3d i %3.3d j %3.3d rho %lf mx %lf
my %lf E %lf p %lf\n", rank, n, i, j, rho[j][i], mx[j][i], my[j][i], E[j][i],
p[j][i]);}
            //printf("rank %d calculating pressure terms for state variables
at i %d j %d\n", rank, i, j);
      }
    }
    if(rank==0&&symmetry_check==1){
    ierr = 0;
```

```
    for (j = 1; j <= mysize+2; j++) {
      for (i = 0; i <= j-1; i++) {
            //printf("Checking cell iter %d i %d j %d\n",n,i,j);
        if (fabs(rho[j][i] - rho[i+1][j-1])>1E-6){
          ierr = 1;
          printf("iter %d Cell %d %d not symmetric with cell %d %d \n rho %f
%f\n",n,i,j,j-1,i+1,rho[j][i],rho[i+1][j-1]);
          printf(" deltarho %f %f\n",rho[j-1][i],rho[i][j-1]);
          if(i!=0&&i!=mysize+1&&j!=1&&j!=mysize+2){
            printf(" rhoplusx left %f %f\n rhoplusx right %f
%f\n",rhoplusx[j-2][i-1],rhoplusy[i-1][j-2],rhoplusx[j-2][i],rhoplusy[i][j-
2]);
            printf(" rhoplusy up %f %f\n rhoplusy down %f %f\n",rhoplusy[j-
2][i-1],rhoplusx[i-1][j-2],rhoplusy[j-1][i-1],rhoplusx[i-1][j-1]);
          }
        }
        if (fabs(mx[j][i] - my[i+1][j-1])>1E-6){
          ierr = 1;
          printf("iter %d Cell %d %d not symmetric with cell %d %d \n mx %f
%f\n",n,i,j,j-1,i+1,mx[j][i],my[i+1][j-1]);
          printf(" deltamx %f %f\n",mx[j-1][i],my[i][j-1]);
          if(i!=0&&i!=mysize+1&&j!=1&&j!=mysize+2){
            printf(" mxplusx left %f %f\n mxplusx right %f %f\n",mxplusx[j-
2][i-1],myplusy[i-1][j-2],mxplusx[j-2][i],myplusy[i][j-2]);
            printf(" mxplusy up %f %f\n mxplusy down %f %f\n",myplusy[j-2][i-
1],mxplusx[i-1][j-2],myplusy[j-1][i-1],mxplusx[i-1][j-1]);
          }
        }
        if(fabs(my[j][i] - mx[i+1][j-1])>1E-6){
          ierr = 1;
          printf("iter %d Cell %d %d not symmetric with cell %d %d \n my %f
%f\n",n,i,j,j-1,i+1,my[j][i],mx[i+1][j-1]);
          printf(" deltamy %f %f\n",my[j-1][i],mx[i][j-1]);
          if(i!=0&&i!=mysize+1&&j!=1&&j!=mysize+2){
            printf(" myplusx left %f %f\n myplusx right %f %f\n",myplusx[j-
2][i-1],mxplusy[i-1][j-2],myplusx[j-2][i],mxplusy[i][j-2]);
            printf(" myplusy up %f %f\n myplusy down %f %f\n",mxplusy[j-2][i-
1],myplusx[i-1][j-2],mxplusy[j-1][i-1],myplusx[i-1][j-1]);
          }
        }
        if(fabs(E[j][i] - E[i+1][j-1])>1E-6){
          ierr = 1;
          printf("iter %d Cell %d %d not symmetric with cell %d %d \n E %f
%f\n",n,i,j,j-1,i+1,E[j][i],E[i+1][j-1]);
          printf(" deltaE %f %f\n",E[j-1][i],E[i][j-1]);
          if(i!=0&&i!=mysize+1&&j!=1&&j!=mysize+2){
            printf(" Eplusx left %f %f\n Eplusx right %f %f\n",Eplusx[j-2][i-
1],Eplusy[i-1][j-2],Eplusx[j-2][i],Eplusy[i][j-2]);
            printf(" Eplusy up %f %f\n Eplusy down %f %f\n",Eplusy[j-2][i-
1],Eplusx[i-1][j-2],Eplusy[j-1][i-1],Eplusx[i-1][j-1]);
          }
        }
      }
    }
    if (ierr == 1) exit(0);
    }
    //*/
```

```
    /*if(rank==0){type filter text
    for(j=2;j<=mysize+3;j++){
        for(i=1;i<=matrix_size_x;i++){
      printf("rank %d n %3.3d i %3.3d j %3.3d rho %lf mx %lf my %lf E %lf p
%lf\n", rank, n, i, j, rho[j][i], mx[j][i], my[j][i], E[j][i], p[j][i]);
        }
    }
    printf("\n");}//*/
    if(rank==0&&debug==1){printf("Done calculations\n");}
    //set temp to ____ and display
    if(n==79&&print_file==1){
      sprintf(filename, "cyclenum%3.3d", n);
      if(rank==0)
        if ((fp=fopen(filename, "w"))==NULL){
              printf("Error: can't open file %s\n", filename);
        }
    }
    for(j=1;j<=mysize+2;j++){
      for(i=0;i<=matrix_size_x+1;i++){
            //Choose what information to display
            temp[j-1][i]=p[j][i];maxScale=2.0;desc="pressure";//pressure
            //temp[j-1][i]=rho[j][i];maxScale=5.0;desc="density";//density
            //temp[j-1][i]=E[j][i];maxScale=10.0;desc="energy";//energy
            //temp[j-
1][i]=0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]);maxScale=0.01;desc="kinetic
energy";//kinetic energy
            //temp[j-1][i]=(E[j][i]-
0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]))/rho[j][i];maxScale=5.0;desc="int
ernal energy";//internal energy

//if(rank==0&&(rho[j][i]<=0.0||E[j][i]<=0.0||p[j][i]<=0.0)){printf("iteration
%d i %3.3d j %3.3d rho %lf mx %lf my %lf E %lf p %lf\n", n, i, j, rho[j][i],
mx[j][i], my[j][i], E[j][i], p[j][i]);}
      }
    }
    /*if(time>=2.5){
            display(matrix_size_x, matrix_size_y, temp, my_offset, mysize,
maxScale);
      printf("Time=%f\n", time);
      exit(0);
    }//*/
    set_label(desc);
    if(n==79&&print_file==1){
      if(rank==0){
        for(i=1;i<=matrix_size_x;i++)
          fprintf(fp, "%f %f\n",i, temp[2][i]);//change 0 to the row you wish
to write to the file
        fclose(fp);
        printf("Wrote to file: %s, iteration:%d\n", filename, n);
      }
      exit(0);
    }
    TotalMass=0.0;
    TotalEnergy=0.0;
    for(j=2;j<=mysize+1;j++){
      for(i=1;i<=matrix_size_x;i++){
            /*if (isnan(rho[j][i])) {
```

```
              printf("Error -- rho[%d][%d]=%f\n",i,j,rho[j][i]);
            }
        if (isnan(E[j][i])) {
              printf("Error -- E[%d][%d]=%f\n",i,j,E[j][i]);
            }//*/
            myTM+=rho[j][i];
        myTE+=E[j][i];
      }
    }
    //MPI_Allreduce(&myTM, &TotalMass, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
    //MPI_Allreduce(&myTE, &TotalEnergy, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
    if(n==0){
        oldTM=TotalMass;
        oldTE=TotalEnergy;
    }

    if(((fabs(TotalMass-oldTM)>1.0E-9)||(fabs(TotalEnergy-oldTE)>1.0E-
9)||isnan(TotalMass)||isnan(TotalEnergy))&&debug==0&&check==1){
       printf("Conservation of mass or Conservation of energy violated\nMass
difference:%e\nEnergy difference:%e\n", TotalMass-oldTM, TotalEnergy-oldTE);
       printf("Problem occured on iteration %5.5d at time %f.\n", n, time);
       exit(0);
    }
      //print iteration info
    if(n%1==0||n==ntimes-1){
      if(rank==0&&(n%1000==0||n==ntimes-1||debug==1)){
        printf("Iteration:%5.5d, Time:%f, Timestep:%f Total mass:%f Total
energy:%f\n", n, time, deltaT, TotalMass, TotalEnergy);
      }
      if(display_on==1){display(matrix_size_x, matrix_size_y, temp,
my_offset, mysize, maxScale);
      /*get_cor(&i, &j, my_offset, matrix_size_x, matrix_size_y);
      if(i>0&&j>0&&j<=mysize){
            E[j][i]+=100.0;
            oldTE  +=100.0;
      }//*/
      }
      //sleep(0);
    /*display time*/
    }
    //else{printf("          %5.5d       %f           %f\n", n, time,
deltaT);}
  }

  /* Return the average time taken/processor */
  slavetime = MPI_Wtime() - starttime;
  MPI_Reduce (&slavetime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
  return (totaltime/(double)size);
}
/*****************************************************************/
double
qxcalc(int i, int j, int NX)
{
  double rdenom, rminus, rplus;
  double duminus1, duplus1, duhalf1;
```

```
  double duminus2, duplus2, duhalf2;
  double duminus3, duplus3, duhalf3;
  double duminus4, duplus4, duhalf4;
  double rnumplus, rnumminus;
  //printf("Entering qxcalc i %d  j %d \n",i,j);
  //  printf("mx[j][i] %f mx[j][i-1] %f\n",mx[j][i],mx[j][i-1]);
  if (i > 1) {
    //printf("Got here\n");
    duminus1 = rho[j][i] - rho[j][i - 1];
    duminus2 = mx[j][i] - mx[j][i - 1];
    duminus3=0.0;//duminus3 = my[j][i] - my[j][i - 1];
    duminus4 = E[j][i] - E[j][i - 1];
  }
  else {
    duminus1 = 0.0;
    duminus2 = 0.0;
    duminus3 = 0.0;
    duminus4 = 0.0;
  }
  //printf("duminus1 %f duminus2 %f duminus3 %f duminus4 %f\n",duminus1,
duminus2, duminus3, duminus4);
  if (i < NX) {
    duplus1 = rho[j][i + 2] - rho[j][i + 1];
    duplus2 = mx[j][i + 2] - mx[j][i + 1];
    duplus3=0.0;//duplus3 = my[j][i + 2] - my[j][i + 1];
    duplus4 = E[j][i + 2] - E[j][i + 1];
  }
  else {
    duplus1 = 0.0;
    duplus2 = 0.0;
    duplus3 = 0.0;
    duplus4 = 0.0;
  }
  duhalf1 = rho[j][i + 1] - rho[j][i];
  duhalf2 = mx[j][i + 1] - mx[j][i];
  duhalf3=0.0;//duhalf3 = my[j][i + 1] - my[j][i];
  duhalf4 = E[j][i + 1] - E[j][i];
  //printf("duhalf1 %f duhalf2 %f duhalf3 %f duhalf4 %f\n",duhalf1, duhalf2,
duhalf3, duhalf4);
  //printf("duplus1 %f duplus2 %f duplus3 %f duplus4 %f\n",duplus1, duplus2,
duplus3, duplus4);
  rdenom = SQ(duhalf1) + SQ(duhalf2) + SQ(duhalf3) + SQ(duhalf4);
  rnumplus =
    duplus1 * duhalf1 + duplus2 * duhalf2 + duplus3 * duhalf3 +
    duplus4 * duhalf4;
  rnumminus =
    duminus1 * duhalf1 + duminus2 * duhalf2 + duminus3 * duhalf3 +
    duminus4 * duhalf4;
  if (rdenom != 0.0) {
    rplus = rnumplus / rdenom;
    rminus = rnumminus / rdenom;
  }
  else {
    rplus = (1.0e-30 + rnumplus) / (rdenom + 1.0e-30);
    rminus = (1.0e-30 + rnumminus) / (rdenom + 1.0e-30);
  }
```

```
  //printf("Exiting qxcalc i %d j %d with rminus %f rplus
%f\n",i,j,rminus,rplus);
  return (qlimit(rminus, rplus));
}

/************************************************************************/
double
qycalc(int i, int j, int JMAX)
{
  double rdenom, rminus, rplus;
  double duminus1, duplus1, duhalf1;
  double duminus2, duplus2, duhalf2;
  double duminus3, duplus3, duhalf3;
  double duminus4, duplus4, duhalf4;
  double rnumplus, rnumminus;
  if (j > 2) {
    duminus1 = rho[j][i] - rho[j - 1][i];
    duminus2 = 0.0;// = mx[j][i] - mx[j - 1][i];
    duminus3 = my[j][i] - my[j - 1][i];
    duminus4 = E[j][i] - E[j - 1][i];
  }
  else {
    duminus1 = 0.0;
    duminus2 = 0.0;
    duminus3 = 0.0;
    duminus4 = 0.0;
  }
  if (j < JMAX+1) {
    duplus1 = rho[j + 2][i] - rho[j + 1][i];
    duplus2 = 0.0;//mx[j + 2][i] - mx[j + 1][i];
    duplus3 = my[j + 2][i] - my[j + 1][i];
    duplus4 = E[j + 2][i] - E[j + 1][i];
  }
  else {
    duplus1 = 0.0;
    duplus2 = 0.0;
    duplus3 = 0.0;
    duplus4 = 0.0;
  }
  duhalf1 = rho[j + 1][i] - rho[j][i];
  duhalf2 = 0.0;//mx[j + 1][i] - mx[j][i];
  duhalf3 = my[j + 1][i] - my[j][i];
  duhalf4 = E[j + 1][i] - E[j][i];
  rdenom = SQ(duhalf1) + SQ(duhalf2) + SQ(duhalf3) + SQ(duhalf4);
  rnumplus =
    duplus1 * duhalf1 + duplus2 * duhalf2 + duplus3 * duhalf3 +
    duplus4 * duhalf4;
  rnumminus =
    duminus1 * duhalf1 + duminus2 * duhalf2 + duminus3 * duhalf3 +
    duminus4 * duhalf4;
  if (rdenom != 0.0) {
    rplus = rnumplus / rdenom;
    rminus = rnumminus / rdenom;
  }
  else {
    rplus = (1.0e-30 + rnumplus) / (rdenom + 1.0e-30);
    rminus = (1.0e-30 + rnumminus) / (rdenom + 1.0e-30);
```

```
  }
  //printf("Exiting qycalc i %d j %d with rminus %f rplus
%f\n",i,j,rminus,rplus);
  return (qlimit(rminus, rplus));
}

/*****************************************************************/
double
qa(double rminus, double rplus)
{
  return (max(min(1.0, rminus), 0.0) + max(min(1.0, rplus), 0.0) - 1.0);
}

/***********************deltamx[j][i]=0.0;*********************************
***********/
double
qb(double rminus, double rplus)//minmod?
{
  return (max(MIN3(1.0, rminus, rplus), 0.0));
}

/*****************************************************************/
double
qc(double rminus, double rplus)
{
  return (max
        (min
         (min(2.0, 2.0 * rminus),
          min(2.0 * rplus, 0.5 * (rminus + rplus))), 0.0));
}

/*****************************************************************/
double
qd(double rminus, double rplus)
{
  return (MAX3(min(1.0, 2.0 * rminus), min(rminus, 2.0), 0.0)
        + MAX3(min(1.0, 2.0 * rplus), min(rplus, 2.0), 0.0) - 1.0);
}

/*****************************************************************/
double
qe(double rminus, double rplus)
{
  double rminabs, rplusabs, q;
  rminabs = fabs(rminus);
  rplusabs = fabs(rplus);
  q = (rminus + rminabs) / (1.0 + rminabs)
    + (rplus + rplusabs) / (1.0 + rplusabs) - 1.0;
  return (q);
}
```

### Display functions

#include <mpi.h>

```
#define MPE_GRAPHICS
#include "mpe.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "hydro.h"

/*int width=1000;
int height=400;//*/
int width=500;
int height=500;//*/
MPE_XGraph graph;
MPE_Color *color_array;
int ncolors=256;
char *label;
void display_init(char *displayname, int iwidth, int iheight){

  int ierr;
  int rank;
  //int ncolors2;

 /* Open the graphics display */

// width=iwidth;
// height=iheight;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
  MPE_Open_graphics( &graph, MPI_COMM_WORLD, displayname,
                     -1, -1, width, height+10, 0 );

  color_array = (MPE_Color *) malloc(sizeof(MPE_Color)*ncolors);

  ierr = MPE_Make_color_array(graph, ncolors, color_array);
  if (ierr && rank == 0) printf("Error(Make_color_array): ierr is %d\n",ierr);

  //MPE_Num_colors(graph, &ncolors2);
  //printf("size of color array is %d\n",ncolors2);

}

 void display_close(void){
   MPE_Close_graphics(&graph);
 }

void display_cond(int matrix_size_x, int matrix_size_y, double **temp,
   int my_offset, int mysize, double maxscale, char *text)
{
  int i, j;
  unsigned int plot_value;
  int xcor, ycor, button;
  /*double localmax=0;
  double localmin=2000;
  for(j=0;j<=mysize;j++){
    for(i=0;i<=matrix_size_x;i++){
```

```
            if(temp[j][i]>localmax)
              localmax=temp[j][i];
            if(temp[j][i]<localmin)
              localmin=temp[j][i];
          }
        }
        printf("localmax %f localmin %f\n", localmax, localmin);//*/
        for(j=1;j<=mysize;j++){
              for(i=1;i<=matrix_size_x;i++){
               int xloc, yloc, xwid, ywid;
                xloc = ((i - 1) * width) / matrix_size_x;
                yloc = ((my_offset + j - 1) * height) / matrix_size_y;
                xwid = (i * width) / matrix_size_x - xloc;
                ywid = ((my_offset + j) * height) / matrix_size_y - yloc;
                plot_value = ncolors - ((double)ncolors*temp[j][i]/maxscale) + 2;
                //printf("temp[%d][%d]=%lf\n",i,j,temp[j][i]);
                if (plot_value < 2) plot_value = 3;
                if (plot_value > ncolors) plot_value = ncolors;
                //printf("%d %d %d %8.5f\n",i,j,plot_value,temp[i][j]);
                if(isnan(temp[j][i]))MPE_Fill_rectangle( graph, xloc, yloc, xwid,
ywid, MPE_WHITE);
                else if(temp[j][i]<0)MPE_Fill_rectangle( graph, xloc, yloc, xwid,
ywid, MPE_WHITE);
                else if(temp[j][i]>maxscale)MPE_Fill_rectangle( graph, xloc, yloc,
xwid, ywid, MPE_BLACK);
                else MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid,
color_array[plot_value]);
```

```
   }
  }
  MPE_Fill_rectangle(graph, 0, height, width, height, MPE_WHITE);
  MPE_Draw_string( graph, 0, height+10, MPE_BLACK, text);
  MPE_Update( graph );
  //MPE_Get_mouse_press(graph, &xcor, &ycor, &button);
  sleep(5);
}

void set_label(char *text)
{
        label=text;
}
void display(int matrix_size_x, int matrix_size_y, double **temp,
   int my_offset, int mysize, double maxscale)
{
  int i, j;
  unsigned int plot_value;
  /*double localmax=0;
  double localmin=2000;
  for(j=0;j<=mysize;j++){
    for(i=0;i<=matrix_size_x;i++){
      if(temp[j][i]>localmax)
        localmax=temp[j][i];
      if(temp[j][i]<localmin)
        localmin=temp[j][i];
    }
```

```
        }
        printf("localmax %f localmin %f\n", localmax, localmin);//*/
        for(j=1;j<=mysize;j++){
              for(i=1;i<=matrix_size_x;i++){
               int xloc, yloc, xwid, ywid;
                xloc = ((i - 1) * width) / matrix_size_x;
                yloc = ((my_offset + j - 1) * height) / matrix_size_y;
                xwid = (i * width) / matrix_size_x - xloc;
                ywid = ((my_offset + j) * height) / matrix_size_y - yloc;
                plot_value = ncolors - ((double)ncolors*temp[j][i]/maxscale) + 2;
                //printf("temp[%d][%d]=%lf\n",i,j,temp[j][i]);
                if (plot_value < 2) plot_value = 2;
                if (plot_value > ncolors) plot_value = ncolors;
                //printf("%d %d %d %8.5f\n",i,j,plot_value,temp[i][j]);
                if(isnan(temp[j][i]))MPE_Fill_rectangle( graph, xloc, yloc, xwid,
ywid, MPE_WHITE);
                else if(temp[j][i]<0)MPE_Fill_rectangle( graph, xloc, yloc, xwid,
ywid, MPE_WHITE);
                else if(temp[j][i]>maxscale)MPE_Fill_rectangle( graph, xloc, yloc,
xwid, ywid, MPE_BLACK);
                else MPE_Fill_rectangle(graph, xloc, yloc, xwid, ywid,
color_array[plot_value]);
             }
        }//*/
        /*//if(debug==1){printf("Color display complete\n");}
        MPE_Fill_rectangle(graph, 0, height/2, width, height/2, MPE_WHITE);
        MPE_Fill_rectangle(graph, 0, height/2, width, 2, MPE_BLACK);
```

```
        for(i=1;i<=matrix_size_x;i++){
              int xloc, xwid;
                xloc = ((i - 1) * width) / matrix_size_x;
                xwid = (i * width) / matrix_size_x - xloc;
                MPE_Fill_rectangle( graph, xloc, height-
((height/2.1)*(temp[1][i]/maxscale)), xwid, 1, color_array[1]);
        }//*/
        //if(debug==1){printf("Graph display complete\n");}
        MPE_Fill_rectangle(graph, 0, height, width, height, MPE_WHITE);
        if(my_offset==0)MPE_Draw_string( graph, 0, height+10, MPE_BLACK,
label);
        MPE_Update( graph );
        sleep(wait_time);
      }
      void display_one_d(int matrix_size_x, int matrix_size_y, double **temp,
        int my_offset, int mysize, double maxscale)
      {
        int i, j;
        unsigned int plot_value;
        /*double localmax=0;
        double localmin=2000;
        for(j=0;j<=mysize;j++){
          for(i=0;i<=matrix_size_x;i++){
            if(temp[j][i]>localmax)
              localmax=temp[j][i];
            if(temp[j][i]<localmin)
              localmin=temp[j][i];
```

```
  }
 }
 printf("localmax %f localmin %f\n", localmax, localmin);//*/
 for(j=1;j<=mysize;j++){
        for(i=1;i<=matrix_size_x;i++){
         int xloc, yloc, xwid, ywid;
          xloc = ((i - 1) * width) / matrix_size_x;
          yloc = ((my_offset + j - 1) * height/2) / matrix_size_y;
          xwid = (i * width) / matrix_size_x - xloc;
          ywid = ((my_offset + j) * height/2) / matrix_size_y - yloc;
          plot_value = ncolors - ((double)ncolors*temp[j][i]/maxscale) + 2;
          //printf("temp[%d][%d]=%lf\n",i,j,temp[j][i]);
          if (plot_value < 2) plot_value = 2;
          if (plot_value > ncolors) plot_value = ncolors;
          //printf("%d %d %d %8.5f\n",i,j,plot_value,temp[i][j]);
          if(isnan(temp[j][i]))MPE_Fill_rectangle( graph, xloc, yloc, xwid,
ywid, MPE_WHITE);
          else if(temp[j][i]<0)MPE_Fill_rectangle( graph, xloc, yloc, xwid,
ywid, MPE_WHITE);
          else if(temp[j][i]>maxscale)MPE_Fill_rectangle( graph, xloc, yloc,
xwid, ywid, MPE_BLACK);
          else MPE_Fill_rectangle(graph, xloc, yloc, xwid, ywid,
color_array[plot_value]);
         }
 }//*/
 //if(debug==1){printf("Color display complete\n");}
 MPE_Fill_rectangle(graph, 0, height/2, width, height/2, MPE_WHITE);
```

```
        MPE_Fill_rectangle(graph, 0, height/2, width, 2, MPE_BLACK);
        for(i=1;i<=matrix_size_x;i++){
              int xloc, xwid;
                xloc = ((i - 1) * width) / matrix_size_x;
                xwid = (i * width) / matrix_size_x - xloc;
                MPE_Fill_rectangle( graph, xloc, height-
((height/2.1)*(temp[1][i]/maxscale)), xwid, 1, color_array[1]);
        }//*/
        //if(debug==1){printf("Graph display complete\n");}
        MPE_Fill_rectangle(graph, 0, height, width, height, MPE_WHITE);
        if(my_offset==0)MPE_Draw_string( graph, 0, height+10, MPE_BLACK,
label);
        MPE_Update( graph );
        sleep(wait_time);
      }
      void get_cor(int *i, int *j, int my_offset, int matrix_size_x, int
matrix_size_y){
              int xcor, ycor, button, wasPressed;
              MPE_Iget_mouse_press(graph, &xcor, &ycor, &button,
&wasPressed);
              if(wasPressed==1&&button==1){
                *i=xcor*matrix_size_x/width+1;
                *j=ycor*matrix_size_y/height+1-my_offset;
              }
              else
              {
                *i=-1;
```

```
                *j=-1;
            }
}
```