# Modeling Spacecraft Reentry

New Mexico

Supercomputing Challenge

Final Report

April 2, 2008

Team Number 56

Los Alamos High School


**Team Members:**

Jonathan Robey
Aric Holland
Lindy Jacobs
Dov Shlachter

**Sponsoring Teacher:**

Diane Medford

**Project Mentor:**

Bob Robey

# Table of Contents

# Executive Summary

- Simulation tool stable to 100km
- CFD linked to Heat Transfer
- Realtime Display/User Interface

We have successfully demonstrated that this simulation, specifically the compressible fluid dynamics (CFD) module is stable to 100km of altitude. However, in these high altitudes the velocity also contributes to the stability envelope.

We have also successfully linked compressible fluid dynamics to heat transfer by exporting data from the model to an excel spreadsheet. It is now at the point that the equations for calculating heat transfer could, with some additional time, be written into the code rather than done separately. This link allows us to experiment with a choice of heat transfer equations to discover which most successfully shows details about the conditions of a spacecraft engaging in reentry; necessary for the creation of a useful model.

A realtime display and user interface has been linked into the main model. This functionality allows the user to receive immediate feedback on the state of the model and permits the user to adjust the conditions of the model more easily.

# 1 Introduction

## *1.1 Problem Statement*

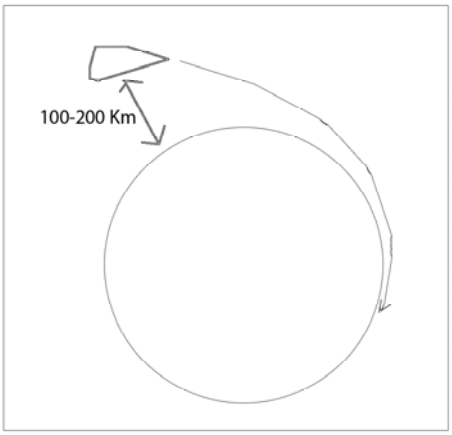
Our project improved an 2-dimensional Eulerian model of an ideal compressible fluid (First written by Jonathan Robey and Dov Shlachter in last years challenge[6]). This model was expanded so it could be used to simulate the reentry of a spacecraft into an atmosphere of known characteristics. Using the model to test various hypotheses about ideal angle and velocity of entry and design of spacecraft, it was hoped that a spacecraft ideally shaped to enter Earth's atmosphere with a minimum of heat-related stress could be designed.Variables considered included angle of entry, velocity of entry, the ratio of the specific heats of the gas and density of air at the edge of the atmosphere, and the specific heat capacity of the material used for the craft.

## *1.2 Objective*

The objective is to create a useful program for modeling spacecraft reentry using compressible fluid dynamics (CFD) and simple heat transfer. In this case CFD refers to gases only with air as the specific gas in the model.

## *1.3 Background*

There are some basic aspects of reentry that are important for all types of spacecrafts (see diagram below). For an acccurate model, it is important to know how large the craft is, which velocities and angles of entry will lead to ideal entry conditions for that craft, how much fuel is needed to slow down the spacecraft to attain the correct velocity, and what materials are used for the heat shield.




The size and shape of the spacecraft are important because they determine how the spacecraft will react when encountering and passing through the atmosphere, such as how much it will slow down and if it is likely to skip off the outer edge of the atmosphere if the velocity is not exactly right. These characteristics are important for determining the negative change in velocity, and whether the craft is likely to 'skip off' the atmosphere if the velocity not ideal. While these variables are important, there are also more universal guidelines for velocity and angle of reentry that all spacecraft must follow. Our model assumed a reentry profile similar to the Apollo mission spaceships and used a velocity and angle accordingly.

For the base code, the model created by this team for last year's competition was used[6], which implemented the two-step Lax-Wendroff method[5] and the three basic Euler equations of fluid dynamics[4], and added multiple features and several layers of complexity to it. The simpler program from last year simulates the behavior of an ideal gas; the basic change to the current computational model was adding the spacecraft as an immobile island with the gas flowing around it. Other changes included the implementation of a simple user interface (UI) and variable boundary conditions.

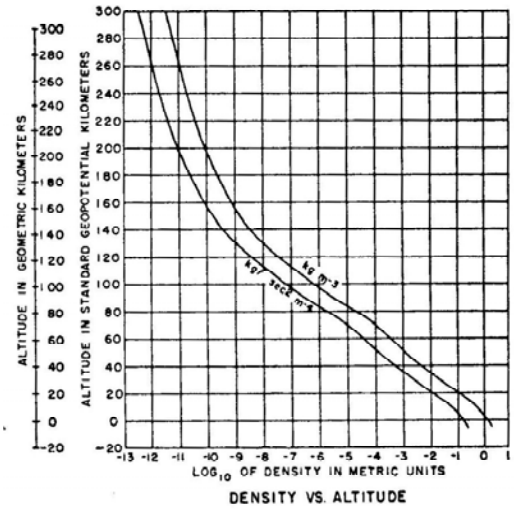
### *1.4 Research*

For better understanding of the problem, several papers on topics similar to ours were investigated. A few books dealing with atmospheric properties, heat transfer, and mathematical equations describing them were studied as well. These sources helped clarify the project and its goals, as well as providing certain variables that are required for calculation. All of the atmospheric properties needed were obtained from The Handbook of Chemistry and Physics[2], a book containing various graphs showing properties of the atmosphere at different altitudes (graphs used are shown below).

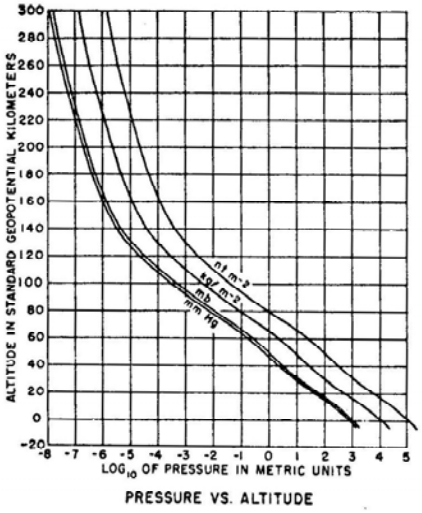
## Standard Atmosphere Graphs

Fluid properties must model the atmosphere--the variation is 7 orders of magnitude! 6 for pressure.

Density Vs Altitude




Pressure Vs Altitude




The papers provided details regarding spacecraft reentry profiles, such as entry angle and velocity, as well as helping to further define the problem.[3, 7, 8, 9, 10] From the papers, it was realized that an important aspect of the problem was to

figure out how the spacecraft would heat up as it reentered. As this was not included in the main model, a spreadsheet was developed using equations from Principles of Heat Transfer[1] to calculate heat fluxes on the spacecraft using time and temperature values obtained from the model. A high school physics textbook[16] was also used at this point to find the density and specific heat of aluminum, as well as to convert several values to the System International.

### *1.5 Innovation*

Some unique features were included in the final version of our model. One of these, the realtime display and graphical user interface (GUI), was already present in an early form in the first year's effort. This display was then improved integrating a larger user interaction capability and also more flexibility in the display. This improvement provides immediate feedback to the user of the code while the model is running, removing the necessity for post-processing on early test runs. More possible uses will almost certainly be developed. Other improvements included the integration of the island configuration file generator, an obvious improvement after the integrated display and GUI was programmed. This particular version of the model is in the preliminary stages of linking a CFD code and a heat transfer model.

# 2 Description

## *2.1 Mathematical Model*

### 2.1.1 Euler Equations

The Euler equations are used to describe the movement of an ideal gas based on the conservation of mass, momentum, and energy[4]. The first three equations (with an additional equation for momentum in each new dimension) state that the change in mass, momentum and energy with time, after accounting for the amount moving across the boundaries of the cells, is zero. The sub-t term in each is the state variable, and the sub-x term is referred to as the flux term. The fourth equation is necessary to be able to calculate the values of the state variables in the next time step because it defines the relationship of pressure to the other state variables. The equation of state being used is based on the Ideal Gas Law.

$$\rho_t + (\rho v)_x = 0$$

[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]

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

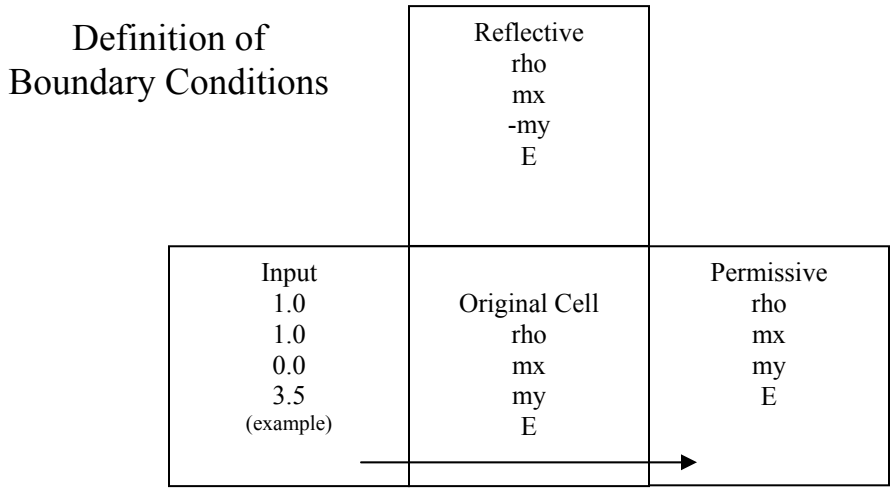
[Equation of State]

### 2.1.2 Boundary Conditions

The model currently has three different types of boundary conditions programmed: reflective, permissive, and input. The reflective boundary condition acts as a solid object just outside the modeled area, preventing any transfer of mass, momentum or energy outside the mesh but not interfering with movement parallel to its surface (slip condition). The permissive boundary condition is a constant interpolation from the cell adjacent to the boundary inside the area being modeled, allowing but not encouraging transfer of mass momentum and energy outside the mesh. The input boundary uses constant state variables along the boundaries to provide various effects, such as one modeling a wind-tunnel.

### 2.1.3 Islands

Islands are rigid, reflective objects placed in the simulation to allow the simulation of air flow in or around solid objects with a more complex structure than a simple box around them. Islands act using the same principles as reflective boundaries inside the simulated area, but due to the possibility of multiple cells bordering on a single island, their implementation is much more difficult.



### 2.1.4 Heat Transfer

Heat transfer is simply the movement of thermal energy from a warmer body to a cooler one. It is important to consider it in the problem of spacecraft reentry for several reasons, the main one being that the heat different materials will attain when passing through the atmosphere determines what materials are usable when building or covering a spacecraft. The hotter a material gets, the likelier it is that it will fail in some way, resulting in a damaged spacecraft that may not make it back to the surface. Because of this, the material being used when creating a spacecraft is always a major factor of concern, and scientists who work with space shuttles are always testing materials to find the ones that hold up the best under the conditions of reentry and undergo the least amount of heat transfer. As heat transfer is such a key component of successful spacecraft reentry and is not included in the code, it is being calculated indirectly using values from the code and fundamental constants.

There are three types of heat transfer: conduction, convection, and radiation. However, in this problem only convection and radiation are being considered. Conduction is ignored because it is the heat transfer between cells within the spacecraft. Because in this model the details of the spacecraft are not known, it is not calculated. Its general effect would be to cool the nose of the spacecraft slightly by distributing the heat throughout the entire structure.

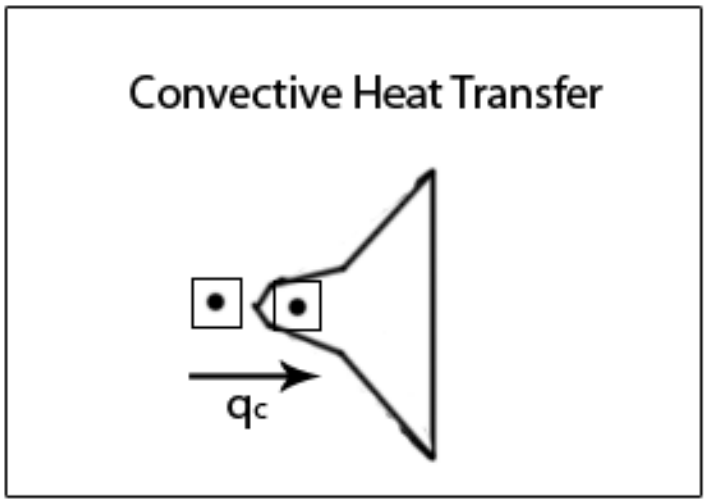
**Conservation of Energy**

No matter how the energy propogate, the net energy remains the same. Shown in the equation:

$$E_t+q_x=[E]_t+[q]_x=0$$

**Convection**

This is the transfer of heat through fluids or gases. In terms of the spacecraft it is the heat that the spacecraft gains from the surrounding air. To calculate convection two temperatures are found at a given point in time: the heat of a cell located at the tip of the spacecraft, and the heat of a cell located just off the tip. This second cell is far enough away from the craft that any forces such as friction do not affect the temperature, but near enough so that the value is accurate in regard to where the spacecraft is in the atmosphere and is not distorted by the shockwave created by reentry. The location of these two cells is illustrated in the diagram below.




The equation used to calculate the heat transfer due to convection is: $q_c=h_cA\Delta T$

**Radiation**

Radiation is the heat transfer that occurs between two objects that are not touching. The spacecraft reentering the atmosphere is losing heat in all directions through radiation. The equation to calculate the heat transfer due to radiation is:

$q_r=\sigma A(T_{sp}^4-T_{cell}^4)$

## *2.2 Computational Model*

### 2.2.1 Basic Model

The basic structure of the model is based on the two-step Lax-Wendroff scheme[5] using total variation diminishing (TVD)[3] to correct for the oscillations that occur before and after a sharp shock. The general concept of Lax-Wendroff is to use the values of the state variables at the current timestep to approximate state values at the interfaces between cells half a timestep ahead, and then use those values to create a better approximation for the full timestep flux terms.

**Lax-Wendroff**

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

First (Predictor) Step

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

Second (Corrector) Step

**TVD**

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

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

TVD terms (added to end of second step)

$$\nu = \frac{\lambda\Delta t}{\Delta x} \qquad \phi(r^+,r^-) = \text{Flux limiter}$$

Definition of functions and variables used in the TVD terms

This method works very well except for at sharp shocks, where a state variable changes rapidly. In these areas the method fails by producing oscillations

in the value before and after the shock. This error is corrected by using the Total Variation Diminishing (TVD) scheme, which works by essentially switching the model to first order near a shock, detecting a shock on the basis of the changes in the slopes of the surrounding cells.

One of the design decisions when creating this model was to use only symmetric methods rather than upwind ones to ease programming the one dimensional version of the model and to avoid some problems inherent in the use of upwind methods in a simulation that has more than one dimension.

### 2.2.2 Boundary Conditions

The boundary conditions are rather easy to set because any boundary cell borders on only one cell within the area being modeled, allowing the boundaries to be set once and then left alone.

Due to the decision to base the boundary conditions on only the cell just inside the boundary, there are a few errors with the non-reflective boundaries. For the permissive boundary, the cell blindly imitates the cell it is adjacent to, removing accurate treatment of shocks that cross it both by producing a small amount of reflection and by not permitting the normal drop in pressure after a shock. The input boundary has a different set of problems in that it never changes the values from the internally defined values for that particular boundary, overwriting any shock waves that reach it and completely ignoring anything happening in the modeled area.

### 2.2.3 Islands

Due to the possibility of multiple cells bordering on one island, the code for working with islands is slightly more complex. During the first step, they are manipulated to duplicate the relationship used for the reflective boundary, but, as there will be multiple cells next to one island, this alone does not solve the

problem. In order to prevent problems during the second step, the model has been programmed to skip certain portions of that step if one of the cells that will be affected is an island. This solution, while relatively simple, works perfectly when tested and duplicates the effect of the more complex solution that uses the same tactic as in the first step.

### 2.2.4 Post-Processing

Heat transfer was calculated in steps and using a spreadsheet. To start, values of time and temperature of the surroundings were imported from the model. From there, three steps were followed:

1. Calculate $q_c$ and $q_r$

For $q_c$ the equation is:

$$q_c=h_cA\Delta T$$

$h_c$=100 W/m$^2$K

A=1 m$^2$

$\Delta T=(T_{Cell}-T_{Spacecraft})$

For $q_r$ the equation is:

$$q_r=\sigma A(T_{sp}^4-T_{cell}^4)$$

$\sigma$=5.67*10^-8

A=1 m$^2$

2. Calculate the energy of the spacecraft (in kJ)

The equation is:

$$E=E_0+\Delta t(q_c+ q_r)$$

$E_0= C_vT$

$C_v$=900 J/kgK

3. Use the calculated energy to find the temperature for the next iteration

Using again the equation:

$E_0= C_vT$

Calculating in the Spreadsheet:

| 1 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 2 | Iteration | Time | $Temp._{cell}$ | $Temp._{spacecraft}$ | $q_c$ | $q_r$ | Energy |
| 3 | 1 | From model | From model | Assumed to be 273.14 | =100*1*(C3-D3) | =5.67*10^-8*1*(D2^4-0) | Calculated separately |
| 4 | 2 | | | =G3/(0.9*2.699) | | | =G2+(E2+F2)*(B3-B2) |

Spreadsheet Sample:

| Iteration | Time (s) | $Temp._{Cell}$ (K) | $Temp._{Sp}$ (K) | $q_c$ (W) | $q_r$ (W) | E (kJ) |
|---|---|---|---|---|---|---|
| 1 | 0.000455 | 346.050091 | 270 | 7605.0091 | 301.327047 | 663.48 |
| | 0.000555 | 495.518352 | 273.4636835 | 22205.46685 | 317.0893797 | 664.2706336 |
| | 0.000655 | 495.518352 | 274.3908811 | 22112.74709 | 321.4117547 | 666.5228892 |
| | 0.000755 | 495.518352 | 275.3144396 | 22020.39124 | 325.760946 | 668.7663051 |
| | 0.000855 | 495.518352 | 276.234375 | 21928.3977 | 330.1368068 | 671.0009203 |
| | 0.000955 | 495.518352 | 277.1507035 | 21836.76485 | 334.5391892 | 673.2267738 |
| | 0.001055 | 495.518352 | 278.0634409 | 21745.49111 | 338.967944 | 675.4439042 |
| | 0.001155 | 495.518352 | 278.9726031 | 21654.57489 | 343.4229212 | 677.6523501 |
| | 0.001255 | 495.518352 | 279.8782059 | 21564.01461 | 347.9039694 | 679.8521499 |
| | 0.001355 | 495.518352 | 280.780265 | 21473.8087 | 352.4109363 | 682.0433417 |
| 2 | 0.004957 | 495.518352 | 313.1453563 | 18237.29957 | 545.2139534 | 760.6613849 |
| 3 | 0.009306 | 463.048857 | 346.7730996 | 11627.57574 | 819.9071247 | 842.3465362 |
| 4 | 0.013547 | 461.043678 | 368.5053357 | 9253.834226 | 1045.5822 | 895.136311 |
| 5 | 0.017742 | 454.0327 | 386.2921917 | 6774.050826 | 1262.544742 | 938.3423629 |
| 6 | 0.021846 | 450.341733 | 399.8701376 | 5047.159544 | 1449.635939 | 971.3245512 |
| 7 | 0.025882 | 443.991557 | 410.6646979 | 3332.685908 | 1612.621862 | 997.5456177 |
| 8 | 0.029896 | 438.716726 | 418.8366404 | 1988.008563 | 1744.864586 | 1017.396083 |
| 9 | 0.033899 | 433.830438 | 424.9881744 | 884.2263644 | 1849.653767 | 1032.338774 |
| 10 | 0.0379 | 429.282827 | 429.4911814 | -20.83543987 | 1929.301327 | 1043.277029 |
| 11 | 0.04191 | 425.375951 | 432.6417097 | -726.5758663 | 1986.536749 | 1050.929977 |
| 12 | 0.04593 | 420.948162 | 434.7268617 | -1377.869967 | 2025.111607 | 1055.99502 |
| 13 | 0.049953 | 418.58122 | 435.7988032 | -1721.758318 | 2045.159537 | 1058.598873 |
| 14 | 0.053978 | 417.099237 | 436.3346765 | -1923.543951 | 2055.237303 | 1059.900563 |

Initially it was thought that the heat transfer equations would not have a stability problem. However, we discovered that the spreadsheet has limitations involving the numeric stability of the time steps. The first time and temperature numbers from the model had been collected every ten iterations, and these worked fine. To increase the amount of time through which heat transfer was calculated—13 values at the ten iterations did not quite make it through one second of the supposed spacecraft's reentry—the second set of numbers was taken at every 1000 iterations. The spreadsheet managed to calculate values for the first seven rows or so, then it began giving a NO NUM report: the numbers were jumping too drastically for it to handle. In order to remain stable the spreadsheet needed the time and temperature collected at a smaller number of iterations. Therefore, for the next set of numbers we collected at every ten iterations again: every 1000 had been proven unstable, and from rough estimates every 100 looked to be unstable as well.

## *2.3 Code Development*

### 2.3.1 Languages

The model is written in C using the MPI, MPE, and X window libraries. C is a compiled language providing a much faster run speed, high enough that a realtime display of the data is feasible with nearly fifty thousand cells. The MPI and MPE libraries lower the difficulties of programming a parallel program slightly by offering already programmed functions to pass data between processors and display functions that are programmed to function properly in a multiprocessor environment.

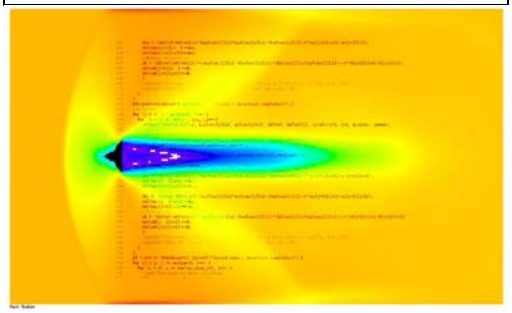
poem.c

```
main(){
//Lines of words and numerals
//Running down the screen
//Many sets of numbers
//Great complexity
//Integers and characters
//Booleans and floats
//Patterns held within
//Chaos shown without
//Mesmerizing motion
//Working to perfect
//Finding the small errors
//Tracking the effect
//Following the thread
//Within the tangled cloth
//Compiling the program
//Setting it to run
return 0;
}
```

-Jonathan Robey

### 2.3.2 Parallel Capability

The core CFD model is completely parallel capable; however, some of the improvements to the model programmed this year have not yet had this capability included. These improvements work very well on one processor and most do not interfere with the multiprocessor capability of the main model. None of the improvements are interfering with the main program's multiprocessor capability but they are not all able to be used in a multiprocessor run without causing errors.

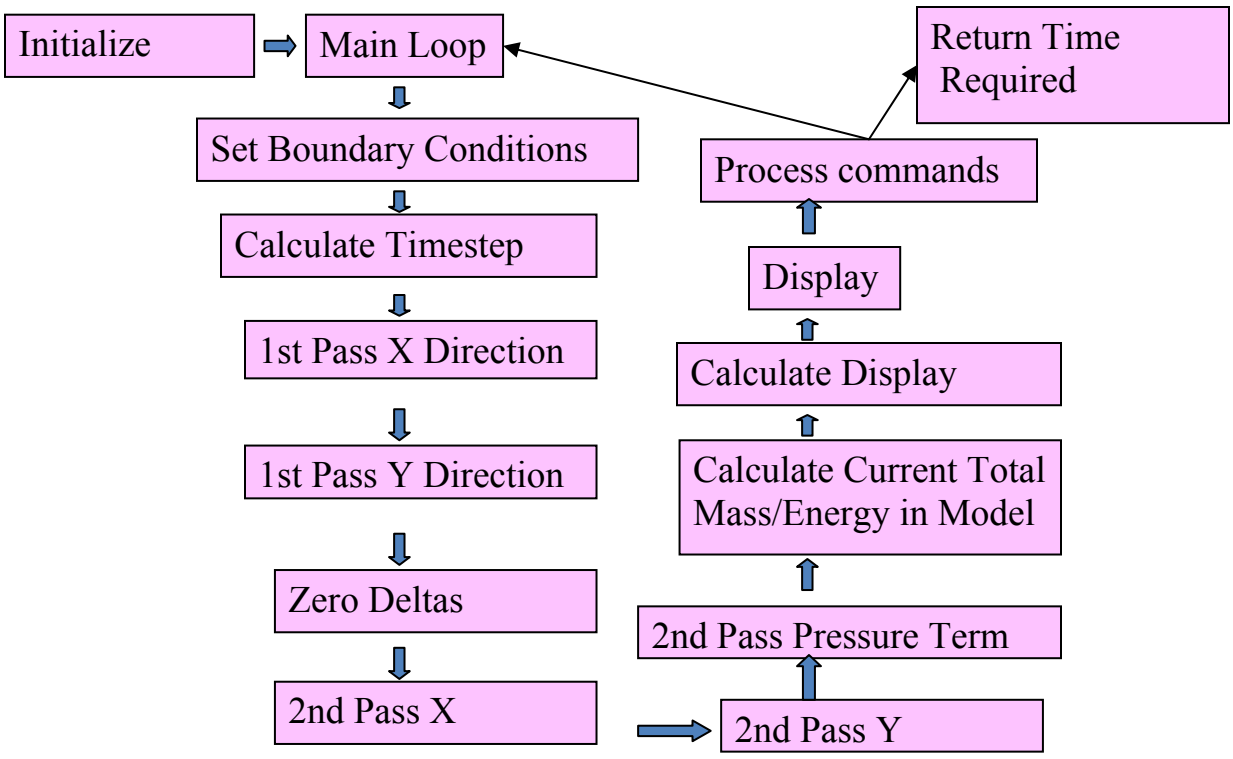


### 2.3.3 Graphical User Interface

One of the improvements to the model is an improved realtime display that is complex enough to be called a GUI. The improvements to the simulation make creating an island configuration file much easier than it would be otherwise; it also helps test any changes to the main model rather quickly.

### 2.3.4 Flow Chart

The basic functional structure of the code is the most sensible arrangement for the nature of the model because of the greater efficiency granted by that structure. The one area in the code that could benefit from a more object-oriented structure is the display functions. The structure used by the display files is already slightly object oriented due to the file structure, but this could be improved on to allow multiple display windows to be open at one time.



### 2.3.5 Lines of Code

The model is very small for its complexity, being less than 2500 lines including the display, initialization, and simulation definition code, as well as main(). The display functions take about 400 of those lines for all the necessary functions to run the display and pass back the mouse and key interactions. Initializations take up another 470 lines including the boundary condition definitions. Header files use up another 90 lines. Most of the rest is the actual model code, including one or two functions split into separate files due the possibility of using them elsewhere as well.

## *2.4 Assumptions and Limitations*

The model is a reasonable approximation of the real world, but as with any approximation there are areas where the model falls short either by assuming the accuracy of another approximation or in ways inherent in the approximation. Many of the following assumptions and limitations are caused by the limits on time available for programming and have known solutions if that time were available.

| Assumptions/Limitations | Explanation |
|---|---|
| Ideal Gas Law | Inaccurate under some conditions, needs a real gas law to improve accuracy |
| 2D model | Overestimates blockage due to assumed infinite depth of plane |
| Non-integration of heat transfer calculations | Slows calculations, does not include transfer of heat energy out of fluid around craft in calculations |
| High momentum, Low density Instability | Does not allow extreme velocities of the spacecraft in upper atmosphere, limits calculation range |
| Regular Mesh, square islands | Prevents the exact shape of the spacecraft from being modeled |

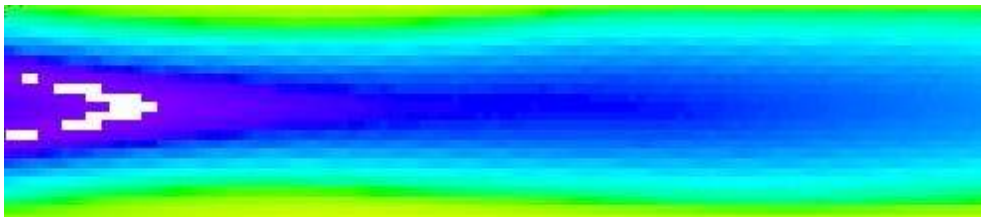
# 3 Results

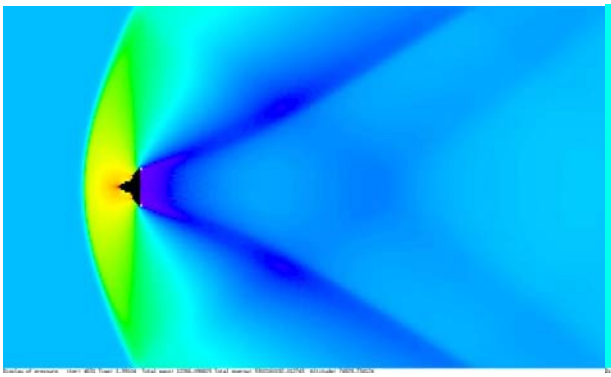
## *3.1 Spacecraft/CFD*

The visual output from the realtime display provides a good description of the main structures of the flow (screenshots on following page). The most prominent of these is the bow shock, a high pressure area just in front of the craft. Another prominent structure is the shadow behind the craft where the fluid recirculates. The third prominent structure is flows off the wingtips that make a roughly 30 degree angle with the shadow of the craft. The current problem is based almost entirely on the temperature of the surrounding fluid, and the obvious maximum at the nose of the craft.

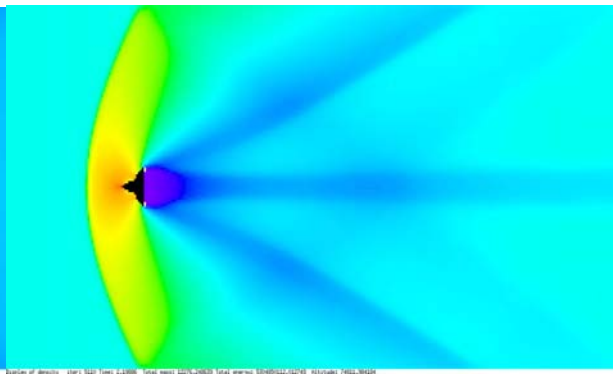
In order to transfer the necessary data to the second part of the model, some of the data from every few iterations must be transferred to a file. In this case the best currently available solution is a comma separated variable file that is written to every ten iterations with the current time and the current internal energy in the cell just off the nose of the craft. There are some difficulties in finding enough time to run a simulation of this length. This problem can be helped by using a coarser mesh, but for an accurate simulation this is not always an option.

As an interesting side note, the model has produced turbulent flow as an emergent property. This means of simulating turbulence is discussed in Implicit Large Eddy Simulation[14]. The thesis of this book is that even without the Navier-Stokes equations or explicit turbulence seeding, certain Eulerian methods are able to simulate turbulence. A screenshot of this effect is included below.

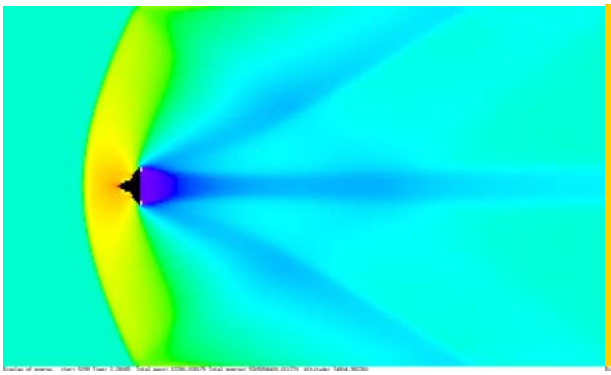




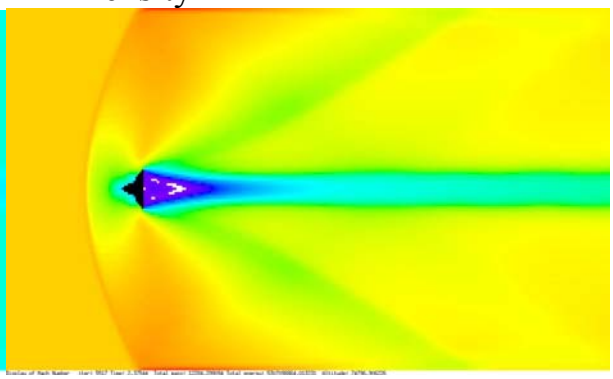
Pressure



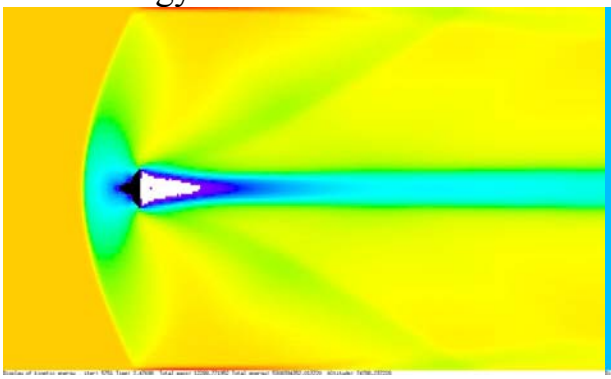
Density



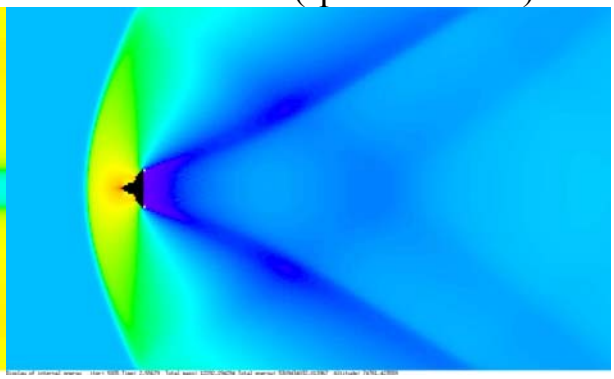
Total energy



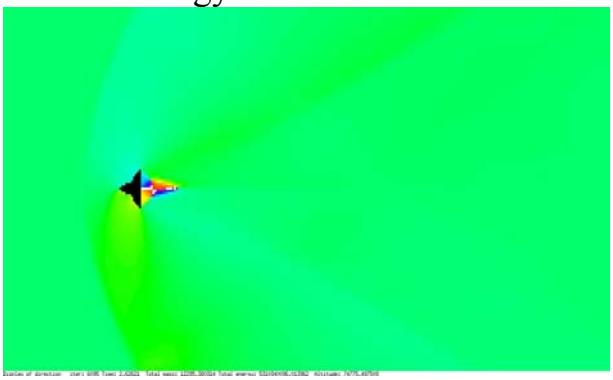
Mach Number(specific to cell)



Kinetic Energy



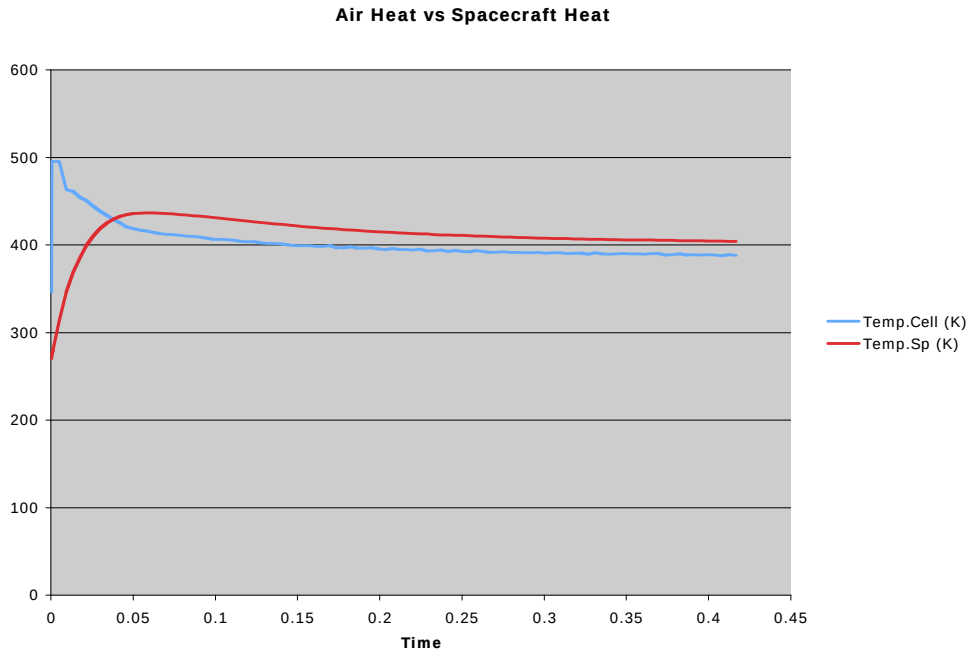
Internal Energy



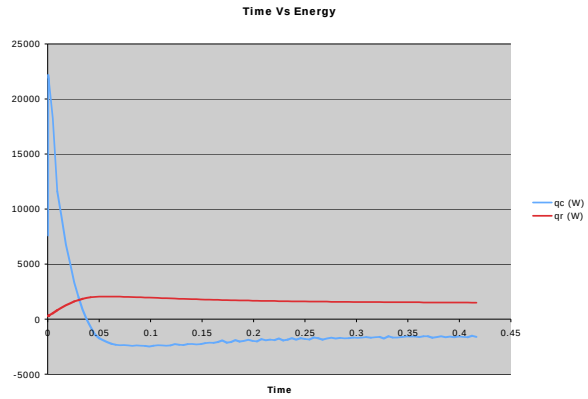
Direction of Fluid flow

### *3.2 Heat Transfer*

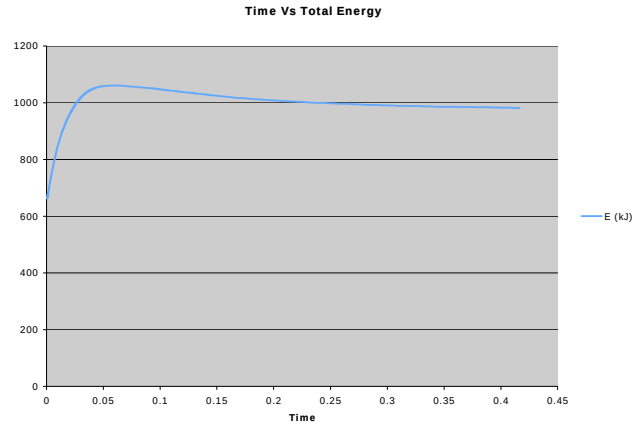
Putting time and temperature values from the code into a spreadsheet to calculate heat transfer allowed us to have a clearer picture of what is happening to the spacecraft as it reenters. It can be seen from the graph (below) that when the spacecraft first hits the atmosphere the outside temperature immediately rises drastically, with the temperature of the spacecraft taking longer to rise. This makes sense, as the spacecraft would take a while to gain heat. Again looking at the graph, it can be seen that the temperature of the spacecraft lags behind the air temperature, eventually—as the temperature of the air stays more constant without much variation—coming close to the temperature of the air. This again makes sense; after the initial contact, the air temperature rises then falls to become nearly constant. The spacecraft takes longer to follow because it takes a while for it to gather heat, and then it maintains the heat as the air temperature drops again, finally coming to rest at a nearly constant temperature a bit higher than the surrounding air.



The heat from convection starts high because the heat is flowing from the air into the spacecraft. It drops as the temperature of the spacecraft rises, and finally becomes negative when the air temperature starts dropping again because the spacecraft cannot lose the heat it has gained as fast as the air can. The heat of radiation starts out low, then rises and steadies at a constant rate radiating out from the spacecraft.

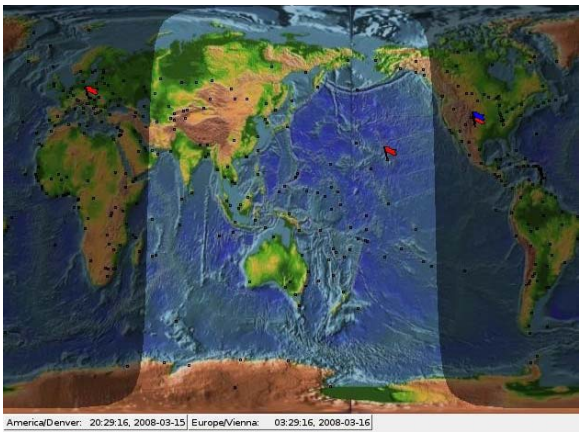



The total energy rises when the spacecraft enters the atmosphere and continues to rise as convection causes the transfer of heat from the air into the spacecraft. It reaches its peak and begins decreasing when the air temperature becomes less than the temperature of the spacecraft, continuing to decrease as the spacecraft loses heat through both convection and radiation.



# 4 Conclusions

## *4.1 Teamwork*




There are four members in our team, two of whom live in Los Alamos, NM, one that resides in Vienna, Austria, and one currently living in Maui, Hawaii. As one may imagine, this complicated matters a bit when it came to collaborating. In our team, Jonathan Robey was the code writer. He wrote all the code himself—a feat those who have not written a code of that length may not appreciate—as well as the sections in the final report having to do with the code. He also helped with other sections of the report. Lindy Jacobs was in charge of heat transfer and writing. She researched spacecraft reentry papers, found equations, and created a spreadsheet to calculate heat transfer. She wrote the sections of the final report dealing with heat transfer as well as the rest of the report with Jonathan's help and the aid of Mr. Robey, the team mentor. Dov Shlachter gave feedback on the report and assisted with writing, editing, and fine-tuning. Aric Holland gave feedback on the report and code.

## *4.2 Model*

### 4.2.1 Analysis

If more time were available to be spent on this model, it would be able to simulate spacecraft reentry reasonably well. Due to the limited time allowed to work on the program, many of these improvements and refinements to the model were not included. These improvements include a three-dimensional model and integration, or at least linkage, to the heat transfer code. Given the necessary time,

however, these difficulties would be handled quite easily other than the few absolute limits set by the necessity for a stable model. One of these limits is the inability of the model to handle high momentum, low density environments when islands are included, excluding high velocity, high altitude conditions from the conditions it is possible to simulate. Other difficulties exist in the lack of details of the structure of the simulated craft; without these details the heat transfer model will be unable to accurately simulate the craft reducing the accuracy of the model.

### 4.2.2 Other Applications

CFD codes are amazingly versatile. The one we developed this year is no exception. As with most other hydrocodes, it is able to calculate the drag force on any given shape, with slightly better results for the stranger shapes than the equations that approximate the drag force. Due to the current 2D nature of the model, this ability is slightly reduced, but this could be corrected given time and incentive to do so. The realtime GUI adds to these inherent capabilities by allowing computational steering or modification of the model's characteristics without recoding the initial conditions and restarting the model.

### 4.2.3 Limitations

Due to both time constraints and inherent qualities of the type of model chosen, there are limitations in what the model can simulate. Most of these limitations were enforced by the amount of time available to work on the model. The model was not tested for convergence and was left as a 2D code for this reason. However, one or two problems were caused by the assumptions in the model. One of the more annoying problems is that the model goes unstable when the mass in each cell is tiny compared to the momentum. This particular limitation forced us to use values smaller by more than an order of magnitude than the reentry profile it was initially based on.

# 5. Recommendations

Given time, there are many areas that we would like to improve. An extremely useful improvement to the CFD code would be to convert it to full three-dimensional capability. This would require a significant amount of work to complete but would certainly be worth it. Another improvement would be to change the equation of state to better reflect reality; currently the equation of state is based on the Ideal Gas Law because that was the only reasonable approximation for the scope of the project. A desired improvement to the specific model discussed in this paper is to either link more closely, or fully integrate the heat transfer model into the CFD code. The current implementation does not allow the CFD code to react to the heat transfer, reducing the accuracy of the model. It would also be useful to look into Adaptive Mesh Refinement and discover whether that would be beneficial in reducing the stairstep approximations to the shape of the spacecraft.

## Acknowledgements

We would like to thank Diane Medford for her role as a mentor and support and Mr. Robey for his role as a mentor and assistance with the model and project specifics. Mr. Robey was extremely helpful with questions concerning both the compressible fluid dynamics code and the heat transfer calculations, as well as proofreading and making suggestions for the final write-up.

# References


[1] Kreith, F. Principles of Heat Transfer. Intext Educational Publishers; New York, New York, 1973. 3rd Edition. Pages 13-14.

[2] Weast, Robert C. et al. Handbook of Chemistry and Physics. 56th Edition. CRC Press; Ohio, 1975.

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

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

[5] Peter Lax and Burt Wendroff, "Systems of Conservation Laws", Communications on Pure and Applied Mathematics, 1960.

[6] Shlachter, Dov and Robey, Jonathan. A Lot of Hot Air: Modeling Compressible Fluid Dynamics. Supercomputing Challenge 2006-7.

[7] Papadopoulos, Dr. Periklis and Subrahmanyam, Prabhakar. "Web-Based Computational Investigation of Aerothermodynamics of Atmospheric Entry Vehicles". *Journal of Spacecraft and Rockets,* Vol. 43, No. 6, November-December 2006.

[8] Fujimoto, Keiichiro and Fujii, Kozo. "Computational Aerodymnamic Analysis of Capsule Configurations Toward the Development of Reusable Rockets". *Journal of Spacecraft and Rockets,* Vol. 43, No. 1, January-February 2006.

[9] Wright, Michael J., Milos, Frank S., and Tran, Philippe. "Afterbody Aeroheating Flight Data for Planetary Probe Thermal Protection System Design". *Journal of Spacecraft and Rockets,* Vol. 43, No. 5, September-October 2006.

[10] Tormo, Vicent Garcia and Serghides, Varnavas C. "Initial Sizing and Reentry-Trajectory Design Methodologies for Dual-Mode-Propulsion Reusable Aerospace Vehicles". *Journal of Spacecraft and Rockets,* Vol. 44, No. 5, September-October 2007.

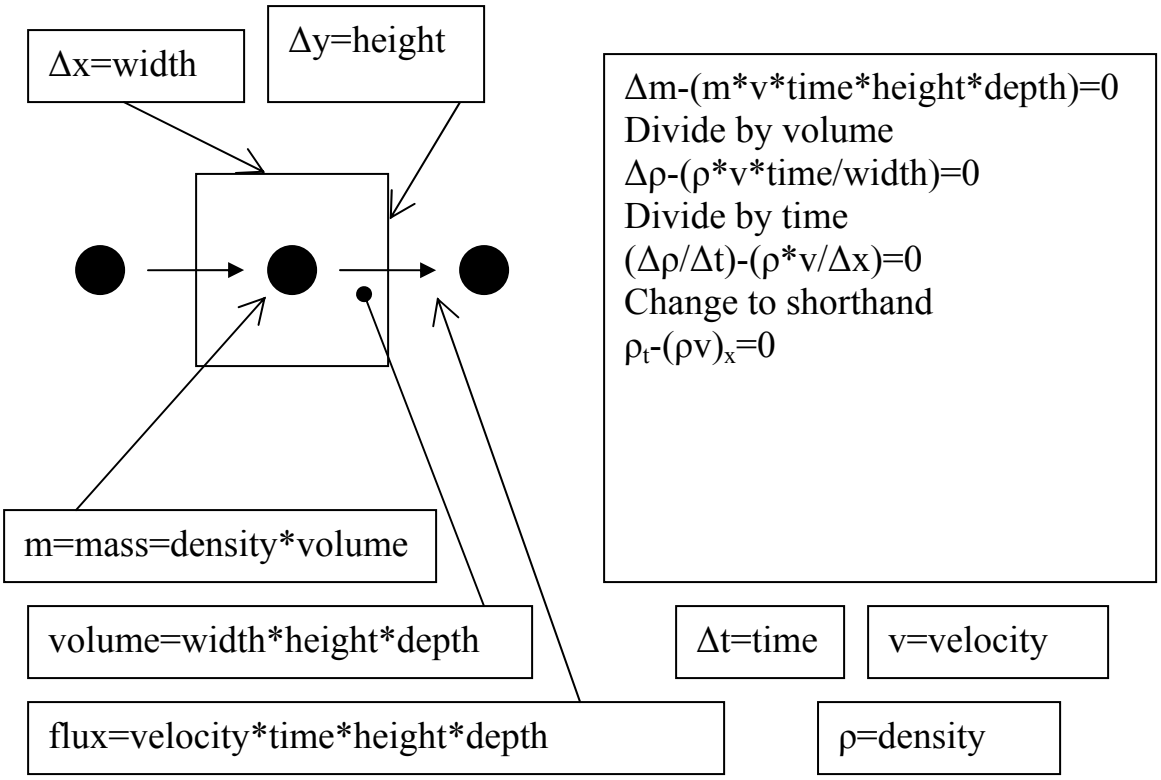
[14] Grinstein,Fernando H. et al. Implicit Large Eddy Simulation. 2007.

[15] "Metals-Specific Heat Capacities." The Engineering Toolbox. http://www.engineeringtoolbox.com/specific-heat-metals-d_152.html. 14 December 2007.

[16] Cutnell, John D. and Johnson, Kenneth W. Physics. Von Hoffmann Press, Inc. 7th Edition.

# Appendices

## *Appendix A: Deriving the Euler Equations*




## *Appendix B: Glossary of Terms*

CFD – Compressible Fluid Dynamics

GUI – Graphical User Interface

2D – Two Dimensional

gamma(γ) – Ratio of specific heats

TVD – Total Variation Diminishing

MPI – Message Passing Interface

MPE – Multi-Processing Environment

## *Appendix C: Code*

Engine:

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

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "hydro.h"
#include "disp.h"
#include "inits.h"
#include "sapient.h"

/*****************************************************************************************
 *                                          SAPIENT
 *                Copyright 2008 Jonathan Robey, Dov Shalchter, Lindy Jacobs, Aric Holland
 *                                Written for Supercomputing Challenge 2007-2008
 *                                        ALL RIGHTS RESERVED
 *                SAPIENT is a proprietary program and is protected by copyright.
 *                  Reproduction and use without permission is expressly prohibited.
 *
 *
*****************************************************************************************/

//define value of PI
#define PI 3.141592653589793238462643
//define macro to find maximum of 4 values
#define MAX4(a,b,c,d) (max(max((a),(b)),max((c),(d))))
//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 4 values
#define MIN4(a,b,c,d) (min(min((a),(b)),min((c),(d))))
//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))
//define sign macro
#define sign(x) (((x)<0)? 1 : -1)

double DMM(double x, double y){  return (.5*(sign(x)+sign(y))*min(fabs(x),fabs(y)));}

double DM4(double w, double x, double y, double z){ return
(0.125*(sign(w)+sign(x))*fabs((sign(w)+sign(y))*(sign(w)+sign(z)))*MIN4(fabs(w),fabs(x),fabs(y),fabs(z)));}

//display functions
void display_init(char *displayname, int iwidth, int iheight);
void display_close(void);
dispFunc display_p;
void display(int matrix_size_y, int matrix_size_x, double **temp, int my_offset,
  int mysize, double maxscale);
void display_adapt(int matrix_size_y, int matrix_size_x, double **temp, int my_offset,
  int mysize, double maxscale);
void display_one_d(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);
```

```
int get_cor(int *i, int *j, int my_offset, int NX, int NY);
char get_key();

//special functions
double getAltitude();
void addValPair(char *filename, double value1, double value2);

//allocation utils
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);

//limiter functions
double noLimiter(double rhoplus, double mplus, double pplus, double deltaT, double deltaD, double q, double
gamma);
double tvd(double rhoplus, double mplus, double pplus, double deltaT, double deltaD, double q, double gamma);

//tvd limiters and qcalcs
double qxcalc(int i, int j, int ISHOCK);
double qycalc(int i, int j, int JMAX);
qlimiter qlimit;
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);

//island utilities
void storeIslands(double **volume, int size_x, int size_y, char *name);
void clearIslands(double **volume, int size_x, int size_y);
void loadIslands(double **volume, char *filename);

//interpolation functions
double average(double p1, double p2, double p3, double p4);
double poly4(double p1, double p2, double p3, double p4);
//double interpMP5(double p1, double p2, double p3, double p4);

//global declarations of state variables and intermediate holders
double   **rho, **mx, **my, **E, **p;       //state variables
double   **rhoplusx, **mxplusx, **myplusx, **Eplusx, **pplusx;//half-step arrays (x direction)
double   **rhoplusy, **mxplusy, **myplusy, **Eplusy, **pplusy;//half-step arrays (y direction)

double engineMP5(sapientInit initCond, MPI_Comm comm, int exampleNum)
{
  char    keyComm;                                                    //characters from keyboard
  int     rank, size ;
  int     next, prev ;
  int     i, j, k;
  int     matrix_size_x, matrix_size_y, ntimes;
  int     mark, markI, markJ, markK;
  int     n;
  int     button;
  int     mysize;
  int     my_offset, edgeT, edgeB;
```

```
  int      maxindX, maxindY, maxindZ, ierr;    //holder variable for index
  int      *counts, *displs;
  int      flag;                          //flag for island
  double   refrho[4], refmx[4], refmy[4], refmz[4], refE[4], refp[4];
  double   *deltaX, *deltaY, **volume;         //size of cell
  double   w;                             //holder var for limiter returns
  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; //timestep variables
  double   vSigma, minRho;                   //variable sigma timestep variables
  double   maxDX, maxDY, maxDZ;               //timestep, sigma, gamma, max for current processor, max for all
  double   maxScale;                         //
  double   time=0;                          //computer simulation time
  double   endTime;                          //time after which to end simulation
  double   slavetime, totaltime, starttime ;   //variables to calculate time taken for the program to run
  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, label[150], info[20];        //variable for labels
  int      dispVar, showDisplay;              //display vars
  char *displayname = ":0";
  //function pointers
  //intitialization functions
  initFunc init;
  //boundry setting function
  boundFunc boundry;
  //updating function for initialization
  updateFunc update;
  //extra info to add to display
  strInfo getInfo;

  //internal function pointers
  sapInterp interp;
  sapLimit limit;



  if(rank==0){printf("Copyright 2007\n");}

  //grab essential values and functins for  the model
  matrix_size_x=initCond.NX;
  matrix_size_y=initCond.NY;
  ntimes=initCond.iters;
  endTime=initCond.time;
  init=initCond.initCond;
  boundry=initCond.boundCond;
  update=initCond.update;
  getInfo=initCond.label;
  dispVar=initCond.dispVar;
  showDisplay=initCond.showDisplay;

  interp=average;
  limit=tvd;
  qlimit=qb;
```

```
  if(matrix_size_y==1) display_p=display_one_d;
  else display_p=display_adapt;

  //initialize special constants
  sigma=initCond.sigma;
  gamma=initCond.gamma;

  mark=initCond.mark;
  markI=initCond.markI;
  markJ=initCond.markJ;
  markK=initCond.markK;

  //check input for acceptability, correct problems if necessary/possible
  if(sigma>1){sigma=0.9;}
  if(dispVar<0||dispVar>NUM_MODES){dispVar=PRESSURE;}
  if(ntimes==-1&&endTime<0&&display_on==0&&update==NULL){printf("Unacceptable end
conditions\n");ntimes=0;} //check for lack of end conditions

  /* 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;

  //initialize display
  if(display_on==1&&showDisplay==1)
   display_init(displayname, 1250, 750);


  /* 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;
  //calculate the upper and lower bounds for the matrix
  edgeT=0;
  edgeB=mysize+3;
  if(debug==1){printf("matrix size x is %d, matrix size y is %d\n", matrix_size_x,matrix_size_y);}
  if(debug==1){printf("my rank is %d, my offset is %d, and mysize is %d\n", rank,my_offset,mysize);}
  if(debug==1){printf("my top row is %d and my bottom row is %d\n", edgeT, edgeB);}
  /* 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*/
//nan preventing init
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;
  }
}
if(rank==0&&debug==1){printf("calling main init function\n");}
//call main init function
init(rho, mx, my, E, deltaX, deltaY, volume, mysize, matrix_size_x, &dispVar, &boundry);
dispVar%=NUM_MODES;
if(rank==0&&debug==1){printf("initial values for state variables set\n");}
//initialize pressure
for(j=0;j<=mysize+1;j++){
  for(i=0;i<=matrix_size_x+3;i++){
    p[j][i]=(gamma-1.0)*(E[j][i]-0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]));
    //printf("i %3.3d j %3.3d rho %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
      if(dispVar==PRESSURE){temp[j-1][i]=p[j][i];maxScale=1.3;desc="initial values of pressure";}
      if(dispVar==DENSITY){temp[j-1][i]=rho[j][i];maxScale=8.0;desc="initial values of density";}
      if(dispVar==ENERGY){temp[j-1][i]=E[j][i];maxScale=5.0;desc="initial values of energy";}
      if(dispVar==MACH_NUMBER){temp[j-
1][i]=sqrt((SQ(mx[j][i])+SQ(my[j][i]))/(gamma*p[j][i]/rho[j][i]));desc="initial values of Mach Number";}
      if(dispVar==KINETIC_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";}
      if(dispVar==INTERNAL_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="initial values of internal energy";}
      if(dispVar==DIRECTION){temp[j-1][i]=(atan2(my[j][i], mx[j][i])+2*PI);maxScale=3*PI;desc="direction";}
            if(volume[j][i]<0){temp[j-1][i]*=-1;}
          }
  }
  if(display_on==1&&showDisplay==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");}
  //prepare to catch key commands

  /* run the simulation for given number of iterations */
  starttime = MPI_Wtime() ;
  for (n = 0; (n < ntimes||ntimes==-1)&&(time<endTime||endTime<0); n++) {
          ierr=0;//no errors this loop so far
    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+11);
    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+14);
    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+17);
    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+25);
   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+27);
   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+30);
   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];
    boundry(&rho[j][0], &mx[j][0], &my[j][0], &E[j][0], 'l', time);

            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];
            boundry(&rho[j][matrix_size_x+1], &mx[j][matrix_size_x+1], &my[j][matrix_size_x+1],
&E[j][matrix_size_x+1], 'r', time);
          }
   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];
            boundry(&rho[1][i], &mx[1][i], &my[1][i], &E[1][i], 't', time);
    }
    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];
     boundry(&rho[mysize+2][i], &mx[mysize+2][i], &my[mysize+2][i], &E[mysize+2][i], 'b', time);
    }
   }
   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;
  maxindX=0;
  maxindY=0;
  minRho=rho[1][1];
  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){
     maxindX=i;
     maxindY=j;
     localmax=hold;
    }
   }
  }
  if(localmax==0){printf("iter %d\n", n); exit(0);}
  MPI_Allreduce(&localmax, &globalmax, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  maxDX=deltaX[maxindX];//temporary solution for constant size of cells, for variable sized cells a line of code is
needed here to find the deltaX for the maximum
  maxDY=deltaY[maxindY];//see line above
        deltaT=sigma/(globalmax/maxDX+globalmax/maxDY);
        time+=deltaT;
        //printf("deltaT %f, maxDX %f, globalmax %f\n", deltaT, maxDX, globalmax);
  if(rank==0&&debug==1){printf("deltaT set to %20.6f\n",deltaT);}

  /* 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++) {
                //find state vars for stencil

                //start from the closest cell and work towards the farthest
                //if one should be treated as an island all of them from there on are treated as islands
                flag=0;//no island encountered
                if(i<0||i>matrix_size_x+1||volume[j+2][i]<0)
                {
                 //if an island
                 refrho[1]=rho[j+2][i+1];
                 refmx[1]=-mx[j+2][i+1];
                 refmy[1]=my[j+2][i+1];
                 refE[1]=E[j+2][i+1];
                 refp[1]=p[j+2][i+1];
                 flag=1;
                }
                else{
                 refrho[1]=rho[j+2][i  ];
                 refmx[1]=mx[j+2][i  ];
                 refmy[1]=my[j+2][i  ];
                 refE[1]=E[j+2][i  ];
                 refp[1]=p[j+2][i  ];
                }
                if(i-1<0||i-1>matrix_size_x+1||volume[j+2][i-1]<0||flag){
                 //if an island
                 refrho[0]=rho[j+2][i+2];
                 refmx[0]=-mx[j+2][i+2];
```

```
                refmy[0]=my[j+2][i+2];
                refE[0]=E[j+2][i+2];
                refp[0]=p[j+2][i+2];
              }
              else{
        refrho[0]=rho[j+2][i-1];
                refmx[0]=mx[j+2][i-1];
                refmy[0]=my[j+2][i-1];
                refE[0]=E[j+2][i-1];
                refp[0]=p[j+2][i-1];
              }
              flag=0;//no island encountered
              if(i+1<0||i+1>matrix_size_x+1||volume[j+2][i+1]<0)
              {
                //if an island
                //to ensure correct treatment of 2 adjacent islands use already set reference cells
                refrho[2]=refrho[1];
                refmx[2]=-refmx[1];
                refmy[2]=refmy[1];
                refE[2]=refE[1];
                refp[2]=refp[1];
                flag=1;
              }
              else{
                refrho[2]=rho[j+2][i+1];
                refmx[2]=mx[j+2][i+1];
                refmy[2]=my[j+2][i+1];
                refE[2]=E[j+2][i+1];
                refp[2]=p[j+2][i+1];
              }
              if(i+2<0||i+2>matrix_size_x+1||volume[j+2][i+2]<0||flag)
              {
                //if an island
                //to ensure correct treatment of 2 adjacent islands use already set reference cells
                refrho[3]=refrho[0];
                refmx[3]=-refmx[0];
                refmy[3]=refmy[0];
                refE[3]=refE[0];
                refp[3]=refp[0];
              }
              else{
                refrho[3]=rho[j+2][i+2];
                refmx[3]=mx[j+2][i+2];
                refmy[3]=my[j+2][i+2];
                refE[3]=E[j+2][i+2];
                refp[3]=p[j+2][i+2];
              }

              //density calculation
//find the interface value
rhoplusx[j][i]=interp(refrho[0],refrho[1],refrho[2],refrho[3]);
//add flux term
        rhoplusx[j][i]-=(deltaT/(2.0*deltaX[i]))*(refmx[2]-refmx[1]);
              //momentum x calculation

              //find the interface value
```

```
                mxplusx[j][i]=interp(refmx[0],refmx[1],refmx[2],refmx[3]);
    //add flux term
            mxplusx[j][i]-=(deltaT/(2.0*deltaX[i]))*
                (((refmx[2]*refmx[2]/refrho[2])+refp[2])
                -((refmx[1]*refmx[1]/refrho[1])+refp[1]));
    //momentum y calculation

    //find the interface value
    myplusx[j][i]=interp(refmy[0],refmy[1],refmy[2],refmy[3]);
    //add flux term
            myplusx[j][i]-=(deltaT/(2.0*deltaX[i]))*
                ((refmx[2]*(refmy[2]/refrho[2]))
                -(refmx[1]*(refmy[1]/refrho[1])));
                    //Energy calculation;

    //find the interface value
    Eplusx[j][i]=interp(refE[0],refE[1],refE[2],refE[3]);
    //add flux term
    //
            Eplusx[j][i]-=(deltaT/(2.0*deltaX[i]))
            *(((refmx[2]/refrho[2])*(refE[2]+refp[2]))
             -((refmx[1]/refrho[1])*(refE[1]+refp[1])));
    //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++){
        //find state vars for stencil

                //start from the closest cell and work towards the farthest
                //if one should be treated as an island all of them from there on are treated as islands
                flag=0;//no island encountered
                if(j+1<edgeT||j+1>edgeB||volume[j+1][i+1]<0)
                {
                  //if an island
                  refrho[1]=rho[j+2][i+1];
                  refmx[1]=-mx[j+2][i+1];
                  refmy[1]=my[j+2][i+1];
                  refE[1]=E[j+2][i+1];
                  refp[1]=p[j+2][i+1];
                  flag=1;
                }
                else{
                  refrho[1]=rho[j+1][i+1];
                  refmx[1]=mx[j+1][i+1];
                  refmy[1]=my[j+1][i+1];
                  refE[1]=E[j+1][i+1];
                  refp[1]=p[j+1][i+1];
                }
                if(j<edgeT||j>edgeB||volume[j][i+1]<0||flag){
```

```
                //if an islandfor(j=1;j<=mysize+2;j++){
                refrho[0]=rho[j+2][i+1];
                refmx[0]=-mx[j+2][i+1];
                refmy[0]=my[j+2][i+1];
                refE[0]=E[j+2][i+1];
                refp[0]=p[j+2][i+1];
              }
              else{
        refrho[0]=rho[j][i+1];
                refmx[0]=mx[j][i+1];
                refmy[0]=my[j][i+1];
                refE[0]=E[j][i+1];
                refp[0]=p[j][i+1];
              }
              flag=0;//no island encountered
              if(j+2<edgeT||j+2>edgeB||volume[j+2][i+1]<0)
              {
                //if an island
                //to ensure correct treatment of 2 adjacent islands use already set reference cells
                refrho[2]=refrho[1];
                refmx[2]=-refmx[1];
                refmy[2]=refmy[1];
                refE[2]=refE[1];
                refp[2]=refp[1];
                flag=1;
              }
              else{
                refrho[2]=rho[j+2][i+1];
                refmx[2]=mx[j+2][i+1];
                refmy[2]=my[j+2][i+1];
                refE[2]=E[j+2][i+1];
                refp[2]=p[j+2][i+1];
              }
              if(j+3<edgeT||j+3>edgeB||volume[j+3][i+1]<0||flag)
              {
                //if an island
                //to ensure correct treatment of 2 adjacent islands use already set reference cells
                refrho[3]=refrho[0];
                refmx[3]=-refmx[0];
                refmy[3]=refmy[0];
                refE[3]=refE[0];
                refp[3]=refp[0];
              }
              else{
                refrho[3]=rho[j+3][i+1];
                refmx[3]=mx[j+3][i+1];
                refmy[3]=my[j+3][i+1];
                refE[3]=E[j+3][i+1];
                refp[3]=p[j+3][i+2];
              }

              //density calculation
//find the interface value
rhoplusy[j][i]=interp(refrho[0],refrho[1],refrho[2],refrho[3]);
//add flux term
      rhoplusy[j][i]-=(deltaT/(2.0*deltaX[i]))*(refmy[2]-refmy[1]);
```

```
                    //momentum x calculation

                    //find the interface value
      mxplusy[j][i]=interp(refmx[0],refmx[1],refmx[2],refmx[3]);
      //add flux term
            mxplusy[j][i]-=(deltaT/(2.0*deltaX[i]))*
                (((refmy[2]*refmx[2]/refrho[2]))
                -((refmy[1]*refmx[1]/refrho[1])));
      //momentum y calculation

      //find the interface value
      myplusy[j][i]=interp(refmy[0],refmy[1],refmy[2],refmy[3]);
      //add flux term
            myplusy[j][i]-=(deltaT/(2.0*deltaX[i]))*
                ((refmy[2]*(refmy[2]/refrho[2])+refp[2])
                -(refmy[1]*(refmy[1]/refrho[1])+refp[1]));
      //Energy calculation;

      //find the interface value
      Eplusy[j][i]=interp(refE[0],refE[1],refE[2],refE[3]);
      //add flux term
      //
            Eplusy[j][i]-=(deltaT/(2.0*deltaX[i]))
            *(((refmy[2]/refrho[2])*(refE[2]+refp[2]))
             -((refmy[1]/refrho[1])*(refE[1]+refp[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]);
            }
   }
   if(rank==0&&debug==1){printf("First pass complete\n");}
   //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++) {
       w=limit(rhoplusx[j][i], mxplusx[j][i], pplusx[j][i], deltaT, deltaX[i], qxcalc(i, j+2, matrix_size_x), gamma);
       //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);
                    if(volume[j+2][i]>=0&&volume[j+2][i+1]>=0){
                    //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]);
                        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);
                        //printf("drho %lf dmx %lf dmy %lf dE %lf\n", drho, dmx, dmy, dE);
    }
   }
   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++){
          w=limit(rhoplusy[j][i], myplusy[j][i], pplusy[j][i], deltaT, deltaY[j], qycalc(i+1, j+1, mysize), gamma);
          //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);
                        if(volume[j+1][i+1]>=0&&volume[j+2][i+1]>=0){
                        //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]);
                        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);
                        //printf("drho %lf dmx %lf dmy %lf dE %lf\n", drho, dmx, dmy, dE);
    }
   }
   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
```

```
            if(volume[j][i]>=0){
            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+1; j++) {
      for (i = 1; i <= matrix_size_x; i++) {
            if(volume[j][i]>=0){
            //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",deltarho[j-1][i],deltarho[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",deltamx[j-1][i],deltamy[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",deltamy[j-1][i],deltamx[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",deltaE[j-1][i],deltaE[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(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
    for(j=1;j<=mysize+2;j++){
        for(i=0;i<=matrix_size_x+1;i++){
       //Choose what information to display
       if(dispVar==PRESSURE){temp[j-1][i]=p[j][i];maxScale=1.3;desc="pressure";}
       if(dispVar==DENSITY){temp[j-1][i]=rho[j][i];maxScale=8.0;desc="density";}
       if(dispVar==ENERGY){temp[j-1][i]=E[j][i];maxScale=10.0;desc="energy";}
       if(dispVar==MACH_NUMBER){temp[j-
1][i]=sqrt((SQ(mx[j][i])+SQ(my[j][i]))/(gamma*p[j][i]/rho[j][i]));desc="Mach Number";}
       if(dispVar==KINETIC_ENERGY){temp[j-
1][i]=0.5*((SQ(mx[j][i])+SQ(my[j][i]))/rho[j][i]);maxScale=0.01;desc="kinetic energy";}
       if(dispVar==INTERNAL_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="internal energy";}
       if(dispVar==DIRECTION){temp[j-1][i]=(atan2(my[j][i], mx[j][i])+PI);maxScale=3*PI;desc="direction";}
```

```
        //if(rank==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(volume[j][i]<0){temp[j-1][i]*=-1;}
                }
        }

        /*if(time>=2.5){
                display(matrix_size_x, matrix_size_y, temp, my_offset, mysize, maxScale);
                printf("Time=%f\n", time);
                exit(0);
        }//*/

        TotalMass=0.0;
        TotalEnergy=0.0;
        myTM=0.0;
        myTE=0.0;
        flag=0;
        for(j=2;j<=mysize+1;j++){
          for(i=1;i<=matrix_size_x;i++){
                if ((isnan(rho[j][i])||rho[j][i]<0)&&((flag&1)==0)) {
                  printf("rho has nan or negative values\n");
                  flag=flag|1;
                }
                if (isnan(mx[j][i])&&((flag&2)==0)) {
                  printf("mx has nan values\n");
                  flag=flag|2;
                }
                if (isnan(my[j][i])&&((flag&4)==0)) {
                  printf("my has nan values\n");
                  flag=flag|4;
                }
            if ((isnan(E[j][i])||E[j][i]<0)&&((flag&8)==0)) {
              printf("E has nan or negative values\n");
                  flag=flag|8;
                }
                if ((isnan(p[j][i])||p[j][i]<0)&&((flag&16)==0)) {
                  printf("Pressure has nan or negative values\n");
                  flag=flag|16;
                }//*/
                myTM+=rho[j][i]*volume[j][i];
            myTE+=E[j][i];
          }
        }
        if(flag!=0){
          ierr=1;
        }
        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);
      ierr=1;
    }

    flag=0;  //do not end simulation
    if(update!=NULL)flag=update(time);//call update function if it exists and check for simulation ending condition
    if(flag){
            ntimes=n;
    }

    info[0]='\n';
    if(getInfo!=NULL){getInfo(info);}

    if(n%1000==0&&mark>-1&&markI>0&&markJ>0){
      addValPair("mark.dat", time, temp[markJ][markI]/rho[markJ][markI]/1000.0);//specific heat of air is ~1kJ/kg
    }//*/
            //print iteration info
    if(n%1==0||n==ntimes-1){
      if(rank==0&&(n%100==0||n==ntimes-1||debug==1)){
        printf("Iteration:%5.5d, Time:%g, Timestep:%f Total mass:%f Total energy:%f  %s\n", n, time, deltaT,
TotalMass, TotalEnergy, info);
      }
      if(display_on==1&&showDisplay==1){
            sprintf(label, "Display of %s   iter: %d Time: %g  Total mass: %f Total energy: %f  %s", desc, n, time,
TotalMass, TotalEnergy, info);
            set_label(label);
            display_p(matrix_size_x, matrix_size_y, temp, my_offset, mysize, maxScale);
        button=get_cor(&i, &j, my_offset, matrix_size_x, matrix_size_y);
        //mouse based commands
        if(i>0&&j>0&&j<=mysize&&button!=-1){
          if(button==1){rho[j+1][i]+=1.0;E[j+1][i]+=10.0;}
          if(button==2){
            //printf("i %d j %d\n %f\n", i, j, temp[j-1][i]);
          }
          if(button==3){volume[j+1][i]*=-1;}
              if(button==4&&mark==0){
                      dispVar+=NUM_MODES-1;
                      dispVar%=NUM_MODES;
              }
              if(button==5&&mark==0){
                      dispVar+=1;
                      dispVar%=NUM_MODES;
              }
        }
        //key based commands
        keyComm=get_key(&i, &j, my_offset, matrix_size_x, matrix_size_y);
        if(keyComm=='q'){
          ntimes=n;
        }
        if(keyComm=='v'){
          printf("i %d j %d\n %f\n", i, j, temp[j-1][i]);
        }
        if(keyComm=='m'){
          if(mark==0){
```

```
            markI=i;
            markJ=j;
            mark=1;
          }
          else{
            mark=0;
          }
        }
        if(keyComm=='s'){
             //popupDialog("File to save islands to", filename, 20);
             storeIslands(volume, matrix_size_x+2, mysize+4, "islands.isl");
            }
            if(keyComm=='c'){
             clearIslands(volume, matrix_size_x+2, mysize+4);
            }
            if(keyComm=='l'){
             //popupDialog("File to load islands from", filename, 20);
             loadIslands(volume, "islands.isl");
            }//*/
      }
    }
    if (ierr != 0) exit(0);
    if(debug==1){printf("starting next loop\n");}
  }

  //finished with display
  if(display_on==1&&showDisplay==1)
     display_close();
  /* 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);
}

/*********************************************************************/
//interpolation functions (1st pass
double average(double p1, double p2, double p3, double p4){
  return 0.5*(p2+p3);
}
double poly4(double p1, double p2, double p3, double p4){
  return ((-p1+9*p2+9*p3-p4)/16);
}
/*double interpMP5(double p1, double p2, double p3, double p4){
  double   VOR, VMP, DJM1, DJ, DJP1, DM4JPH, DM4JMH, VUL, VAV, VMD, VLC, VMIN, VMAX;//variables
for 5th order method
  double   B1=0.01666666667, B2=(4/3), ALPHA=4, eps=1E-10;//constants needed for 5th order MP5 scheme

  VOR=B1*(2.0*p1-13.0*p2+47.0*p3+27.0*p4-3.0*p5);
  VMP=p3+DMM(p4,ALPHA*(p3-p2));


  if((VOR-p3)*(VOR-VMP)<eps)
  {return VOR;}


  DJM1=p1-2.0*p2+p3;
```

```
 DJ  =p2-2.0*p3+p4;
 DJP1=p3-2.0*p4+p5;

 DM4JPH=DM4(4.0*DJ-DJP1,4.0*DJP1-DJ,DJ,DJP1);
 DM4JMH=DM4(4.0*DJ-DJM1,4.0*DJM1-DJ,DJ,DJM1);

 VUL=p3+ALPHA*(p3-p2);
 VAV=0.5*(p3+p4);
 VMD=VAV-0.5*DM4JPH;
 VLC=p3+0.5*(p3-p2)+B2*DM4JMH;

 VMIN=max(MIN3(p3,p4,VMD),MIN3(p3,VUL,VLC));
 VMAX=min(MAX3(p2,p4,VMD),MAX3(p3,VUL,VLC));

 return VOR+DMM(VMIN-VOR,VMAX-VOR);
}*/


/*********************************************************************/
//limiter functions(2nd pass)
double noLimiter(double rhoplus, double mplus, double pplus, double deltaT, double deltaD, double q, double
gamma){
        return 0.0;
}
double tvd(double rhoplus, double mplus, double pplus, double deltaT, double deltaD, double q, double gamma){
            double   nu, cs, v, cv, w;// nu holders for second pass(TVD) and holder vars for calculations
        v=mplus/rhoplus;
        cs=sqrt(gamma*pplus/rhoplus);
        nu=(fabs(v)+cs)*deltaT/deltaD;
        cv=nu*(1.0-nu);
        w=0.5*cv*(1.0-q);
        return w;
}
/*********************************************************************/
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:
#include <mpi.h>
#define MPE_INTERNAL
#define MPE_GRAPHICS
#include "mpe.h"
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "hydro.h"
//keyboard input files
#include <termios.h>
//#include <l7.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>


/*int width=1000;
int height=400;//*/
int width=750;
int height=750;//*/
MPE_XGraph graph;
XWindowAttributes winAttrib;
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_Update(graph);
  //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;
  /*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_BLACK);
            else if(temp[j][i]>maxscale)MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, color_array[ncolors]);
            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 display_cond_click(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_BLACK);
              else if(temp[j][i]>maxscale)MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, color_array[ncolors]);
              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);
}
```

```
void set_label(char *text)
{
            label=text;
}
void display_adapt(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=200000;
  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&&temp[j][i]>=0)
        localmin=temp[j][i];
    }
  }
  localmin*=.99;
  localmax*=1.01;
  maxscale=localmax-localmin;
  if(maxscale<.000001)
    maxscale=.000001;
  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]-localmin)/maxscale)) + 2;
               //printf("temp[%d][%d]=%lf\n",i,j,temp[j][i]);
               if (plot_value < 3) 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_BLACK);
               else if((temp[j][i]-localmin)<0)MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, color_array[3]);
               else if((temp[j][i]-localmin)>maxscale)MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid,
MPE_WHITE);
               else MPE_Fill_rectangle(graph, xloc, yloc, xwid, ywid, color_array[plot_value]);
    }
  }//*/
  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(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_BLACK);
              else if(temp[j][i]>maxscale)MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, color_array[ncolors]);
              else MPE_Fill_rectangle(graph, xloc, yloc, xwid, ywid, color_array[plot_value]);
   }
  }//*/
  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=/home/jon/workspacetemp[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_BLACK);
            else if(temp[j][i]>maxscale)MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, color_array[ncolors]);
            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, MPE_WHITE);
  MPE_Fill_rectangle(graph, 0, height/2, width, 2, MPE_BLACK);
  //MPE_Fill_rectangle(graph, 0, height, 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);
}
int 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){
              *i=xcor*matrix_size_x/width+1;
              *j=ycor*matrix_size_y/height+1-my_offset;
              return button;
            }
            else
            {
              *i=-1;
              *j=-1;
              return -1;
            }
}

//functions to catch commands from keyboard

char Iget_key_press(int *xpos, int *ypos){
  XEvent event;
  char keys[20], toReturn;
  int numChar;
  KeySym keysym;
  XComposeStatus compose;
```

```
  if (graph->Cookie != MPE_G_COOKIE) {
    fprintf( stderr, "Handle argument is incorrect or corrupted\n" );
    return '\0';
  }

  XSelectInput( graph->xwin->disp, graph->xwin->win, MPE_XEVT_IDLE_MASK | ButtonPressMask |
KeyPressMask );
  /* add mouse presses to the events being monitored */

  if (XCheckWindowEvent( graph->xwin->disp, graph->xwin->win, KeyPressMask,
                                  &event ) == False) {
    return '\0';
  }
  /* will check once if mouse has been pressed */

  numChar=XLookupString(&event, keys, 20, &keysym, &compose);

  toReturn='\0';

  *xpos=event.xkey.x;
  *ypos=event.xkey.y;

  if(((keysym>=XK_KP_Space)&&(keysym<=XK_KP_9))
   ||((keysym>XK_space)&&(keysym<XK_asciitilde))){
            toReturn=keys[0];
   }

  XSelectInput( graph->xwin->disp, graph->xwin->win, MPE_XEVT_IDLE_MASK );
  /* turn off all events */

  return toReturn;
}


char get_key(int *i, int *j, int my_offset, int matrix_size_x, int matrix_size_y){
            char ch;
            int xcor, ycor;


            ch=Iget_key_press(&xcor, &ycor);
            if(ch!='\0'){
              *i=xcor*matrix_size_x/width+1;
              *j=ycor*matrix_size_y/height+1-my_offset;
            }
            MPI_Bcast(&ch, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
            MPI_Bcast(i, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(j, 1, MPI_INT, 0, MPI_COMM_WORLD);
            return ch;
}
```